University of Alberta

CORRECT AND EFFICIENT SEARCH ALGORITHMS IN THE PRESENCE OF REPETITIONS

by

Akihiro Kishimoto ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2005

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

0-494-08259-3

Your file Votre référence
ISBN:
Our file Notre reference
ISBN:

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

AND/OR tree search has been a fundamental topic in Artificial Intelligence, because many tasks can be decomposed into subtasks, such that either all (AND) or one (OR) of them must be solved. Recent AND/OR tree search algorithms have become powerful, especially by using the notion of proof and disproof numbers. However, there are limitations of these algorithms if the search space involves repetitions. Repetitions cause a problem of efficiency versus correctness. Some approaches incorrectly deal with repetitions to preserve search efficiency. As a result, they occasionally return incorrect solutions. Other approaches compromise efficiency to guarantee correctness. However, they are not efficient enough to become satisfactory choices of practitioners.

This thesis presents effective and correct methods for AND/OR tree search with repetitions. The one-eye problem in the game of Go, tsume-Go (life and death problem), and checkers are used as application domains to explore the new techniques.

The thesis contains four research contributions. First of all, a solution to the Graph History Interaction (GHI) problem, which may cause a solver to return the incorrect outcome because of repetitions, is presented. Theoretical and empirical results show that the GHI solution is general, correct, and efficient. Secondly, a performance problem is presented when the depth-first proof number (df-pn) search algorithm, which is an effective algorithm using proof and disproof numbers, is adapted to domains involving repetitions. A solution to the problem is given and dramatical improvements over df-pn are empirically achieved. Thirdly, on top of these solutions, domain dependent enhancements are added to the programs that solve the one-eye and tsume-Go problems. These techniques are very promising, and contribute to surpass the performance of the best existing tsume-Go solver. Finally, a divide and conquer approach that can reduce the search space is presented. This approach further improves the performance of the one-eye solver.

Acknowledgements

It was my pleasure being a graduate student at the University of Alberta. There are so many people that I would like to thank. It would have been impossible to finish my endeavor without them.

First of all, I would like to thank my supervisor, Martin Müller. I will never forget our friendship since we met in Japan and a lot of constructive advice that was always to a point. Jonathan Schaeffer is the professor who made me what I am. If I had not met him at the workshop where Jonathan gave an invited talk, I would have had no opportunity to be a graduate student in Canada. Needless to say, I learned a lot from his suggestions. Additionally, I would like to thank the other committee members, Michael Buro, Robert Hayes, and Keh-Hsun Chen for their valuable feedback.

Thank you to the Computer Go Group for discussing a lot of research topics on Computer Go: Martin Müller, Markus Enzenberger, Adi Botea, Ling Zhao, Xiaozhen Niu, Zhichao Li, and David Silver.

I am very much indebted to Jonathan Schaeffer, Yngvi Björnsson, and Neil Burch. They not only provided code for a checkers program but also gave me beneficial advice. Their comments made the research excellent.

I would like to acknowledge the current and former members of the GAMES group, especially Michael Bowling, Robert Holte, and Nathan Sturtevant, for their helpful feedback.

Yasushi Tanase has not only been working with me as the leader of the ISshogi Project, but also been one of the best friends. His enthusiasm for the game of Shogi made me determined to work on research on games.

I will never forget my friends in Canada, Adi Botea, Viorica Botea, Darse Billings, Maria Cutumisu, and Xiaomeng Wu. Markian Hlynka taught me many useful English expressions for topics, ranging from technical writing to daily conversation.

Thomas Wolf of Brock University generously gave me executable code of his tsume-Go solver GoTools, a problem collection, and comments to a draft of the thesis. Anders Kierulf kindly provided his Go-playing program SmartGo and gave me beneficial suggestions. I appreciate them very much.

Last but not least, special thanks go to my wife, Sarina. Without her superb understanding of my research and deep devotion, I could not have completed the thesis.

Contents

1 Introduction **1**
- 1.1 Artificial Intelligence and Games 1
- 1.2 Problem Solving and Games 2
- 1.3 Target Domains 3
  - 1.3.1 The Tsume-Go Problem in Go 3
  - 1.3.2 The One-Eye Problem in Tsume-Go 6
  - 1.3.3 Checkers 8
- 1.4 Contributions 8
- 1.5 Outline of this Thesis 9
- 1.6 Publications 9

2 Literature Review **11**
- 2.1 Advances in AND/OR Tree Search 11
  - 2.1.1 Terminology for AND/OR Tree Search 12
  - 2.1.2 Standard Search Strategies 13
  - 2.1.3 Depth-First versus Best-First Search for AND/OR Tree Search 15
  - 2.1.4 The AO* Algorithm 15
  - 2.1.5 Proof-Number Search 16
  - 2.1.6 The PN* Algorithm 18
  - 2.1.7 The PDS Algorithm 20
  - 2.1.8 The Df-pn Search Algorithm 21
  - 2.1.9 Simulation 23
  - 2.1.10 The $\alpha\beta$ Algorithm 24
  - 2.1.11 Replacement and Garbage Collection Schemes 26
  - 2.1.12 Summary 27
- 2.2 The GHI Problem 27
  - 2.2.1 Problem Description 27
  - 2.2.2 Palay's Suggestions 29
  - 2.2.3 Campbell's Analysis 30
  - 2.2.4 Breuker's Base-Twin Algorithm 31
  - 2.2.5 Nagai's Approach 33
  - 2.2.6 Other Related Work 34
  - 2.2.7 Summary 36
- 2.3 Previous Research on Computer Go 36

2.3.1 Tsume-Go Solvers 36
2.3.2 Proving Territories Safe 38
2.3.3 Heuristic Estimations of Eyes 40
2.3.4 Other Related Work 41
2.4 Research Issues 42

3 A General Solution to the GHI Problem 44
3.1 The Algorithm to Solve the GHI Problem 44
3.1.1 Overview of the Solution 44
3.1.2 Duplicating Transposition Table Entries 45
3.1.3 Encoding Paths 45
3.1.4 Invoking Simulation for Correctness 46
3.1.5 Reducing Simulation Calls 46
3.1.6 Algorithm-Specific Implementation Details 47
3.2 Correctness of the Solution 48
3.2.1 When Proofs and Disproofs Fit in Memory 48
3.2.2 When Proofs and Disproofs Do Not Fit in Memory . . 50
3.3 Game-Specific Implementation Details 53
3.3.1 Go 53
3.3.2 Checkers 53
3.4 Other Implementation Issues 54
3.5 Experiments 55
3.5.1 Setup 55
3.5.2 Results in Go 56
3.5.3 Results in Checkers 58
3.5.4 Results with Limited Memory 64
3.5.5 An Example of GHI in Go with the SSK Rule 68
3.6 Conclusions 70

4 Depth-First Proof-Number Search with Repetitions 73
4.1 Problem Description 73
4.2 Computing Proof and Disproof Numbers in Domains with Repetitions 75
4.3 Experimental Results 79
4.3.1 Setup 79
4.3.2 Results 80
4.4 Conclusions 85

5 Domain Dependent Knowledge for the One-Eye Problem 87
5.1 The Basic One-Eye Algorithm 87
5.2 Game-Specific Search Enhancements 89
5.2.1 Safety by Connection To Safe Stones 89
5.2.2 Forced Moves 90
5.2.3 Simulation 91
5.2.4 Heuristic Initialization 92

- 5.3 Ko and Ko Threats 96
- 5.4 Empirical Results 98
  - 5.4.1 Setup of Experiments 98
  - 5.4.2 Adding Enhancements 99
  - 5.4.3 Leaving out Enhancements 101
  - 5.4.4 Performance of Simulation 101
  - 5.4.5 Re-searches for Ko 101
  - 5.4.6 Reexpansion of Interior Nodes 103
  - 5.4.7 Currently Unsolved Problems 105
  - 5.4.8 Comparison to Other Solvers 105
- 5.5 Conclusions and Future Work 106

6 Domain Dependent Knowledge for the Tsume-Go Problem 107
- 6.1 The Basic Two-Eye Algorithm 107
- 6.2 Game-Specific Knowledge 110
- 6.3 Experimental Results 114
  - 6.3.1 Setup 114
  - 6.3.2 Performance on Enhancements 115
  - 6.3.3 Comparison with GoTools 117
  - 6.3.4 Comparison with SmartGo 123
- 6.4 Currently Unsolved Problems 123
- 6.5 Conclusions and Future Work 124

7 Divide and Conquer Approach to the One-Eye Problem 125
- 7.1 Basic Idea 125
- 7.2 The Dynamic Decomposition Search Algorithm 127
- 7.3 Search Control 128
- 7.4 Using the Transposition Table in DDS 128
- 7.5 Experimental Results 130
  - 7.5.1 Setup of Experiments 130
  - 7.5.2 Results 130
- 7.6 A Relaxed Decomposition Model 134
- 7.7 Conclusions and Future Work 140

8 Conclusions and Future Work 142
- 8.1 Conclusions 142
- 8.2 Future Work 143

A Proof that Df-pn Loops Forever in the Example of Figure 4.2151

B Glossary 155

C Positions Used for Experiments 158
- C.1 Go 158
  - C.1.1 Positions Used for All Experiments 158
  - C.1.2 Positions Used in Chapter 7 167

C.2 Checkers 170

List of Figures

1.1 A tsume-Go problem (Black to live). 4
1.2 An enclosed tsume-Go problem (Black to Live). 6
1.3 Example of a one-eye problem (Black to live). 7

2.1 Proof and disproof numbers. 17
2.2 Snapshot of expanding a tree in PNS. 18
2.3 Multiple iterative deepening. 19
2.4 Pseudo-code of the df-pn algorithm. 22
2.5 Pseudo-code of simulation. 25
2.6 Kawano's simulation. 26
2.7 The GHI problem. 27
2.8 Prototypical case of GHI. 30
2.9 An example where BTA fails with the current-player-loss scenario. 32
2.10 An example showing why BTA must remove possible-draw marks. 33
2.11 An example of safe stones but an unsafe territory. 38
2.12 An example of Benson's unconditional safety. 39
2.13 An example of safety by locally alternating play. 40

3.1 Figure used to prove that the GHI solution is free from the draw-first and draw-last cases. 49
3.2 An example of constructing an incorrect proof tree with replacement schemes. 51
3.3 Example of a hard problem (Black to live). 56
3.4 Node expansions for problems solved by both HANDLE-GHI and IGNORE-GHI in df-pn in Go. 59
3.5 Execution time for problems solved by both HANDLE-GHI and IGNORE-GHI in df-pn in Go. 59
3.6 Node expansions for problems solved by both HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in Go. 60
3.7 Execution time for problems solved by both HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in Go. 60
3.8 Comparison of node expansions between HANDLE-GHI and IGNORE-GHI in df-pn in checkers. 61
3.9 Node expansions for problems solved by HANDLE-GHI and NAGAI in df-pn in checkers. 62

3.10 Node expansions for problems solved by HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in checkers. 63
3.11 Comparison of the solving ability of DISCARD-PROOFS and KEEP-PROOFS in df-pn. 66
3.12 Comparison of the solving ability of DISCARD-PROOFS and KEEP-PROOFS in $\alpha\beta$. 68
3.13 Repetitions in the move sequences in Figure 3.14. 70
3.14 An instance of the GHI problem in Go. 71

4.1 A problem with repetitions in df-pn. 74
4.2 An example in which df-pn loops forever even with the GHI solution. 75
4.3 Df-pn with minimal distance *md*. 76
4.4 Pseudo-code of the df-pn(r) algorithm. 77
4.5 Updating E's minimal distance. 78
4.6 Computing C's minimal distance. 79
4.7 Node expansions for problems solved by both versions in Go. . 81
4.8 Execution time for problems solved by both versions in Go. . . 81
4.9 Node expansions for problems solved only by df-pn(r) in Go. . 82
4.10 Execution time for problems solved only by df-pn(r) in Go. . . 82
4.11 An example easily solved by df-pn(r) in Go (oneeyed.8.sgf, Black to live at **A2**). 83
4.12 Node expansions for problems solved by both versions in checkers. 84
4.13 Node expansions for problems solved only by df-pn(r) in checkers. 84
4.14 An example easily solved by df-pn(r) in checkers. 85
4.15 An example from checkers showing the problem of computing proof and disproof numbers with repetitions. 86

5.1 An example of seki. 88
5.2 Connections to safe stones. 89
5.3 Connection to safe stones on protected liberty. 89
5.4 Forced moves. 90
5.5 An example showing that simulation is effective. 92
5.6 Heuristic initialization for the one-eye problem. 94
5.7 Example of an indirect move that must be generated. 94
5.8 Pseudo-code of the df-pn algorithm with nonuniform threshold increments. 97
5.9 Performance for 20 hard problems for each enhancement. . . 100
5.10 Comparison of node expansions between UNIT-N and UNIT-1. 104
5.11 Black to play and live: A currently unsolved problem. 105

6.1 Example in which the basic one-eye algorithm fails. 107
6.2 A live position without two eyes. 109
6.3 An example that is not a large safe eye. 110
6.4 Forced moves. 112

6.5 A position in Wolf's test collection. 114
6.6 Comparison of execution time for individual instances in LV6.14. 119
6.7 A position that GoTools solves more quickly than TsumeGo Explorer (P1031246, White lives with **D1**). 119
6.8 A position that TsumeGo Explorer solves more quickly than GoTools (P2072215, Black kills with **E2**). 120
6.9 Execution time for problems solved by both programs in ONEEYE. 121
6.10 A position that is hard for GoTools but very easy for TsumeGo Explorer (oneeyec.10.sgf, Black to kill by playing at **C5**). . 121
6.11 A position that GoTools solves statically (oneeyeb.10.sgf, White to kill by playing at **F9**). 122
6.12 Execution time for problems solved only by TsumeGo Explorer in ONEEYE. 122
6.13 A position solved only by TsumeGo Explorer (oneeyee.9.sgf, Black to live by playing at **D8**). 123

7.1 A position to which a divide and conquer approach is applicable. (Black to play.) 126
7.2 Interacting regions in ko with the divide and conquer approach. 126
7.3 Pseudo-code of the DDS algorithm. 129
7.4 Example of using proof numbers to select the working region. . 129
7.5 Node expansions for toy problems solved by both versions. . . 132
7.6 A position that DDS solved with much less nodes than no-DDS (divide-conquer.12.sgf, Black to live by playing at **O12**). . . . 132
7.7 Execution time for standard problems solved by both versions. 133
7.8 Node expansion for standard problems solved by both versions. 133
7.9 A position that DDS solved more quickly (oneeyee.1.sgf, Black lives with **B2**). 134
7.10 Relaxed decomposition. 135
7.11 Pseudo-code of the RDS algorithm. 136
7.12 Decomposition for tsume-Go. 140

8.1 An example in Igo Hatsuyo-ron (White to live) and its enclosed version. 143

A.1 An example in which df-pn loops forever. 154

B.1 Blocks. 155
B.2 Eye. 156
B.3 Ko. 156
B.4 Liberties. 156
B.5 Seki. 157

List of Tables

3.1 Performance comparison between ignoring and dealing with the GHI problem for df-pn in Go. All statistics are computed for 157 problems solved by both program versions. 57
3.2 Performance comparison for $\alpha\beta$ in Go. All statistics are computed for 138 problems solved by both versions. 57
3.3 Performance comparison for df-pn in checkers. Node statistics are computed for the subset of 160 problems solved by all program versions. 58
3.4 Extra problems solved by each method. See Appendix C.2for a listing of the 200 problems. 61
3.5 Performance comparison for $\alpha\beta$ in checkers. Node statistics are computed for the subset of 119 problems solved by both program versions. 62
3.6 List of problems used for df-pn. 65
3.7 List of problems used for $\alpha\beta$. 65
3.8 List of problems solved by df-pn with a 5 MB transposition table. 67
3.9 List of problems solved by df-pn with a 30 MB transposition table. 67
3.10 List of problems solved by $\alpha\beta$ with a 5 MB transposition table. 69
3.11 List of problems solved by $\alpha\beta$ with a 30 MB transposition table. 69

4.1 Performance comparison between old df-pn and df-pn(r) in Go. 80
4.2 Performance comparison between old df-pn and df-pn(r) in checkers. 80

5.1 Performance for successively switching on enhancements. . . . 99
5.2 Performance for turning off single enhancement. 101
5.3 Performance for turning on single enhancement. 102
5.4 Performance of simulation for all 157 solved problems (all enhancements on, phase 1 searches only). 102
5.5 Overhead for ko re-searches. 103
5.6 Performance comparison between UNIT-N and UNIT-1. 104
5.7 List of unsolved problems. 105

6.1 Performance for successively switching on enhancements in LV6.14.115

6.2 Performance for turning off single enhancement in LV6.14. . . 116
6.3 Performance for turning on single enhancement in LV6.14. . . 116
6.4 Performance data on simulation for all 566 solved problems in LV6.14. All enhancements on. Phase 1 searches only. 116
6.5 Overhead for ko re-searches. 117
6.6 Performance comparison between TsumeGo Explorer and GoTools in LV6.14. 118
6.7 Performance comparison between TsumeGo Explorer and GoTools in ONEEYE. 118
6.8 List of unsolved problems in ONEEYE. 124

7.1 Performance comparison for DDS and no-DDS in Toy. 131
7.2 Performance comparison for DDS and no-DDS in Standard. . 131

Chapter 1

Introduction

1.1 Artificial Intelligence and Games

Since playing games is one of the intellectual behaviors of human beings, games have been described as a vehicle to conduct research in Artificial Intelligence (AI) [23, 72]. There are many reasons why games are ideal domains for exploring the capabilities of AI:

- **Clear motivations:** Games raise an interesting question of whether machines can outperform human players in terms of intelligence. Games therefore supply clear and strong motivations for researchers to defeat the best human players.

- **Simplicity and difficulty:** The rules of games are usually fixed and short. Also, games have clear results that are either wins, draws, or losses. It is therefore easier for researchers to measure improvements than in many real-world applications. Yet despite the simplicity of game rules, a large amount of work and scientific inventions is needed for computers to play well.

- **Many applications:** The techniques investigated in games have often been adapted to real-world domains. For example, search algorithms used in solving puzzles can be applied to path-finding and DNA sequence alignment. As well, some of the techniques developed for games engines easily transfer to other search engines.

Since the dawn of research on programming games, chess has been a major testbed because of its popularity in the western world. McCarthy mentioned that chess is referred to as the *Drosophila* of AI [50]. As the Drosophila, the scientific name for the common fruit fly, is used for biological experiments, chess is similarly used for experiments in AI. A variety of techniques were invented by research in chess, including specialized hardware [8, 19, 27], and search algorithms [47, 64]. Massively parallel search was also an important contribution [24, 27, 69]. These techniques contributed to DEEP BLUE [14], the culmination of a multi-year effort at IBM Research, which beat the World Chess Champion, Garry Kasparov in 1997 [74].

1.2 Problem Solving and Games

Strong game-playing programs such as chess programs employ look-ahead search algorithms to improve their move decisions. One of the crucial procedures in games is to find a winning way from a given position. Since a player has to prove that one of the moves must lead to a win (i.e., OR procedure), as well as that all the moves played by the opponent lead to losses for the opponent (AND procedure), this process can be seen as *AND/OR tree search.*

Two properties are required for a solver that finds a winning way:

- **Efficiency:** The solver must quickly return a solution, since solving hard positions improves the strength of game-playing programs.
- **Correctness:** The solver must be a perfect player. In other words, if the solver returns a result, that result must always be reliable.

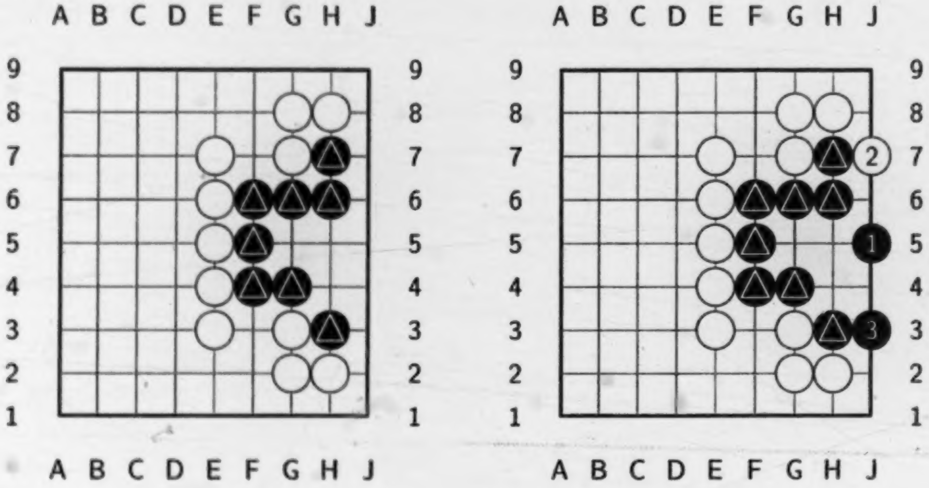
In terms of efficiency, search algorithms using the notion of *proof* and *disproof numbers* have been shown to be effective in many games, such as five-in-a-row, chess, and Shogi (Japanese chess) [4, 2, 10, 56, 77]. In particular, Nagai's depth-first proof-number (df-pn) search is a most promising method [56]. His tsume-shogi (Shogi checkmating problem) solver solved all known difficult problems with solution sequences of 300 moves and more. Proof and disproof numbers are good estimates of the difficulty of finding a solution. One of the important enhancements to these algorithms is a *transposition table*. The transposition table is a cache that keeps previously searched results, making use of the fact that the search space explored in practice is not a tree, but a graph [78]. If more than one path leads to an identical state, and a result that state is stored in the transposition table, a search algorithm can just retrieve a table entry and omit searching below that state, thus saving duplicate effort. However, in terms of correctness, the transposition table may contain flaws, if the search space involves repetitions, since the transposition table does not contain the paths by which the identical positions are reached. This problem is called the *Graph-History Interaction (GHI) Problem* [60]. Programs have suffered from GHI over decades. Programmers either ignore the GHI, since they do not want to degrade the performance of their solvers, or compromise the effectiveness of the solvers to guarantee correctness of the solutions returned by the solvers. The major contribution of this thesis is the design of a high-performance AND/OR tree search algorithm which always guarantees correctness.

1.3 Target Domains

Three domains are chosen as target applications: the *tsume-Go* problem in the game of *Go*, the *one-eye* problem in Go, and the game of *checkers*. The reasons why these three domains are suitable for investigating a research topic of this thesis is explained.

1.3.1 The Tsume-Go Problem in Go

After DEEP BLUE's victory against the human world champion, a larger number of researchers have moved to more complex games such as Go and Shogi. Researchers have started investing significant resources especially to Go, because of its popularity in Asian countries. According to McCarthy, Go can be a new drosophila of AI due to the difficulty of the domain [50]. The search space of chess is estimated between 10^{43} and 10^{50} [2]. 19×19 Go has $3^{361} \approx 10^{172}$



(a) (b)

Figure 1.1: A tsume-Go problem (Black to live).

possible positions of which about 1.2% are legal. The number of 3^{361} for Go ignores move history. If the history is taken into account, the number is much larger. There are two reasons why Go is especially difficult. One essential difference between chess and Go is the number of possible moves in a position (*branching factor*). Since the branching factor in Go is usually larger than in chess, *brute-force* approaches taken by chess programs are not as effective in the domain of Go. The other difficulty in Go is evaluation of positions. In chess, very good and efficiently computable criteria are available such as the material balance. In contrast, so far there are no clear strategies for evaluating positions in Go. Slight differences in the location of stones can completely change the result of a game. As a result, despite a lot of efforts, the best Go programs can still be easily beaten even by human players of moderate skills.

One weakness of Go programs is recognizing whether groups of stones are alive or dead. Such problems are called *tsume-Go (life and death)* problems and play a critical role in deciding the outcome of many games. Figure 1.1 (a) shows an example. Given the crucial stones marked by triangles, the tsume-Go problem is to check whether these stones can live or not. If the crucial stones cannot be captured, the stones are proven to be alive. For example, if the crucial stones can make two eyes*, they are alive (see an example of a life-making sequence for Black in Figure 1.1 (b)). If the eye space is too small, the crucial stones are proven to be dead. Therefore, the process to prove if crucial stones are alive can be conducted as an AND/OR tree search. Although the tsume-Go problem is simpler than the whole game of Go, tsume-Go solvers still suffer from the difficulties of Go. Hence, building a high-performance solver is a good domain for research on effective AND/OR tree search algorithms. Besides, since the search space of tsume-Go involves many repetitions of identical board positions, for example by two consecutive passes, tsume-Go is also an ideal domain to investigate a new solution to the GHI problem.

This thesis focuses on *enclosed* positions, which are easier than open positions. The currently best tsume-Go solver, Wolf's GoTools [90], achieves high performance on such positions with a small to moderate branching factor. However, the state-of-the-art tsume-Go solvers have difficulties in solving tsume-Go problems with a larger branching factor. Therefore, new methods must be invented to tackle such enclosed problems.

This thesis follows Wolf's definitions for enclosed tsume-Go problems. A problem is defined by the following parameters:

- The two players, called the *defender* and the *attacker*. The defender tries to live and the attacker tries to kill.
- The *region*, a subset of the board. At each turn, a player must either make a legal move within the region or pass.
- One or more blocks*[1] of *crucial stones* of the defender. The crucial stones are part of the region. The defender wins a tsume-Go problem by making one of crucial stones safe from capture inside the region, typically creating two eyes connected to one of the crucial stones. The attacker can win by either capturing *all* crucial stones, or by preventing the defender from creating two eyes in the region.

---

[1] See Appendix B for a definition of the Go terms marked by *.

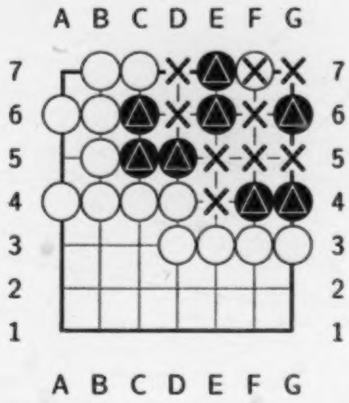


Figure 1.2: An enclosed tsume-Go problem (Black to Live).

- *Safe attacker stones*, which surround the region.
- The player moving first.
- Seki* is a win for the defender. The *situational super-ko (SSK) rule* is used to determine legal moves. It states that any move that repeats a previous board position with the same color to play is illegal.

Figure 1.2, adapted from [90], shows an enclosed tsume-Go problem. The problem is completely closed off by safe white stones. Black is the defender and White is the attacker. Crucial stones are marked by triangles and the rest of the region is marked by crosses. The task for Black is to live inside the area enclosed by the safe stones and for White to prevent that. There is an unsafe stone at **F7**. If this stone is captured, a player might play at such a point later, so they are part of the region.

1.3.2 The One-Eye Problem in Tsume-Go

The *one-eye problem* as another target domain is selected. This problem is a special case of tsume-Go. It addresses the question of whether a player can create an eye connected to the player's stones in a given region. Although this problem is simpler than full tsume-Go, which is concerned with making two

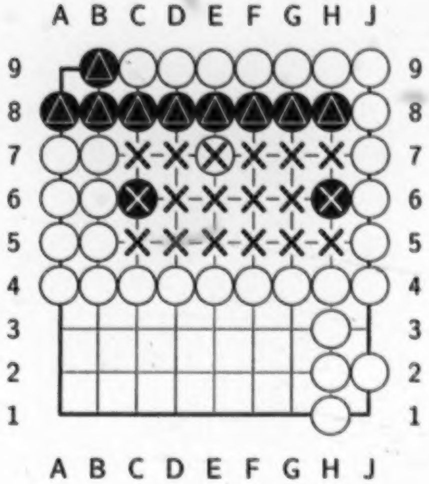


Figure 1.3: Example of a one-eye problem (Black to live).

eyes, there are many similarities. For example, every tsume-Go problem in which the group under attack has already surrounded one eye in some region reduces to the one-eye problem on the rest of the board.

A one-eye problem in a given Go position is defined similarly to enclosed tsume-Go by the defender, attacker, region, crucial stones, and safe attacker stones. Only the winning conditions of the attacker and defender for the one-eye problem is listed:

- The defender wins a one-eye problem by creating an eye connected to all the crucial stones inside the region. The attacker can win by either capturing *at least one* crucial stone, or by preventing the defender from creating a connected eye in the region.

There is a small difference of dealing with crucial stones between enclosed tsume-Go and the one-eye problem. The blocks surrounding the eye cannot be captured.

Figure 1.3 shows an example of a one-eye problem. Black is the defender and White is the attacker. Again, crucial stones are marked by triangles and the rest of the region is marked by crosses. Black must make an eye inside the region. while White tries to prevent that.

1.3.3 Checkers

Compared with Go, checkers is a simpler game for game-playing programs. Checkers has a smaller branching factor and simpler position evaluation. Checkers is estimated to have 5×10^{20} possible positions [73]. It is a game in which brute-force search has been quite successful. CHINOOK was the first game-playing program to win a World Championship [71]. After this milestone in research on CHINOOK, the next goal for checkers has become to solve the game itself. However, despite a lot of effort including large endgame databases [73], the search space of 10^{20} is still challenging. Further, since checkers involves repetitions, it suffers from the GHI problem. Therefore, checkers is an ideal domain to investigate correct and efficient search algorithms.

1.4 Contributions

The contributions of this thesis can be summarized as follows:

- A novel solution to the GHI problem is presented. Compared with the previous solutions, the GHI solution is so general and practical that it is applicable to many games and algorithms. As demonstrated with two applications in checkers and the one-eye problem in Go, the solution incurs only a very small overhead, while guaranteeing correctness.

- Empirical evidence that repetitions cause not only the GHI problem, but also a performance issue for the df-pn algorithm is presented. An improved version of df-pn, called df-pn(r)is developed. Results on the one-eye problem and in checkers show that df-pn(r) improves the solving ability of df-pn.

- One-eye and tsume-Go solvers based on df-pn(r)are developed. By further adding correct domain-dependent knowledge to the solvers, very promising results are achieved. These solver succeeds in solving hard problems that the current best tsume-Go solvers are unable to solve.

- As a further improvement to the one-eye solver, a divide and conquer approach, called *dynamic decomposition search (DDS)* is presented. DDS achieves a reduction of the search space, thereby achieving better performance on average. Additionally, *relaxed decomposition*, a more ambitious way of splitting positions is introduced.

1.5 Outline of this Thesis

Chapter 2 reviews literature on AND/OR tree search algorithms, previous solutions to the GHI problem, and computer Go. Chapter 3 presents a solution to the GHI problem. Chapter 4 addresses a problem of the df-pn algorithm in domains with repetitions and describes a solution. Chapter 5 gives the details of the one-eye solver. Chapter 6 explains techniques for the tsume-Go solver. Chapter 7 presents a divide and conquer approach for the one-eye solver. Chapter 8 concludes the thesis and indicates further research directions.

1.6 Publications

Chapter 3 is based on three papers. Additional unpublished material that deals with the case of replacing proven and disproven transposition table entries is added to the chapter. "A Solution to the GHI Problem for Depth-First Proof-Number Search" was presented at the 7th Joint Conference on Information Sciences (JCIS 2003) [38], and an extended version was accepted for Information Sciences [40]. The later paper "A General Solution to the Graph History Interaction Problem" generalized the method to adapt to various algorithms and games [39]. It was presented at the 19th National Conference on Artificial Intelligence (AAAI'04). The techniques described in the chapter contributed to solving the first standard opening in checkers [68], and is used in an ongoing effort to solve the whole game. They are also incorporated into the tsume-shogi solver of ISshogi [35]. Chapter 4 is based on the paper "Df-pn in Go: Application to the One-Eye Problem" [37], presented at the 10th Advances in Computer Games Conference. The chapter includes additional results in checkers. Chapter 5 is based on the paper "Df-pn in Go: Application to the One-Eye Problem [37]." The chapter improved on the paper by including further enhancements. Chapter 7 is based on the paper that has been accepted by the IEEE Symposium on Computational Intelligence and Games in 2005. Additionally, Chapters 4 to 7 contain material from an invited talk "Search Techniques for the One-Eye and Tsume-Go Problems (in Japanese)" given at the 9th Game Programming Workshop in Japan (GPW 2004) [36].

Chapter 2

Literature Review

This chapter introduces previous approaches on search and computer Go. Section 2.1 gives a survey of AND/OR tree search algorithms. Section 2.2 discusses the GHI problem and previous solutions. Section 2.3 reviews the relevant literature on computer Go. Section 2.4 describes some research questions related to this previous work.

2.1 Advances in AND/OR Tree Search

Since solving tsume-Go problems involves searching AND/OR trees, the methods available for this domain are surveyed. Some definitions for AND/OR tree search are given in Section 2.1.1. Then, before understanding AND/OR tree search algorithms, the basics of two standard search algorithms, *best-first* and *depth-first*, are briefly reviewed in Section 2.1.2. The advantages and disadvantages of these algorithms are explained and enhancements to depth-first search are discussed. Next, two best-first and three depth-first search algorithms for AND/OR trees are introduced from Section 2.1.3 to Section 2.1.8. Their advantages and disadvantages are also discussed. An enhancement to AND/OR tree search is described in Section 2.1.9. The $\alpha\beta$ algorithm, which is the most popular algorithm for two-player game-tree search, is briefly presented in Section 2.1.10. Replacement and garbage collection schemes are described in Section 2.1.11. Finally, concluding remarks are given in Section 2.1.12.

2.1.1 Terminology for AND/OR Tree Search

The terminology for AND/OR trees and their relation to game-tree search are defined in this section. There are two types of nodes: *OR* and *AND* nodes. OR nodes correspond to positions where the first player to play, AND nodes to positions where the opponent moves next. At each OR node only one branch has to be solved, while at each AND node all the branches must be solved. All children of an OR node are AND nodes, and all children of an AND node are OR nodes. The *root* node is the only node that does not have any parents. The root is assumed to be an OR node. Each node can have three kinds of values: **true**, **false**, and **unknown**. Once an AND/OR tree is solved, the value of the root node must be determined to be either **true** or **false**. A *terminal* node is a node that has no children. Hence, when a node is evaluated as a terminal node, the value of that node must be either **true** or **false**. A node having at least one child is called an *interior* node. A *leaf* node is a node that has not been expanded. It is unknown whether a leaf node is terminal or interior.

If at least one of the children of an interior OR node has value **true**, that OR node also has value **true**. If all the children have value **false**, the value of that node is **false**. Similarly, if one of the children of an interior AND node has value **false**, that AND node has value **false**, whereas the AND node has value **true** if all its children have value **true**. A node having value **true** is called a *proven* node, while a node having value **false** is called a *disproven* node.

An AND/OR tree is analyzed for the player who is to play at the root. The player to play at the root is called the *first player*, and the opponent is called the *second player*. Considering the value of an AND/OR tree from a viewpoint of the first player, a node associated with value **true** is called a *win*, and a node with value **false** is a *loss*. A *proof* is also used to express a win, while a *disproof* stands for a loss.

In AND/OR trees, a *proof tree* guarantees that a node is proven. In other words, a proof tree has the following properties:

1. The root node is in the proof tree.

2. For each interior OR node in the proof tree, at least one child of the OR node is in the proof tree.

3. For each interior AND node in the proof tree, all the children of the AND node are in the proof tree.

4. All terminal nodes in the proof tree are proven.

A *disproof tree* which provides a disproof is defined in an analogous way.

2.1.2 Standard Search Strategies

Depth-First Search versus Best-First Search

Depth-first search (DFS) [65] is a search strategy that selects one branch, pursues down that branch until it reaches a terminal node or a certain maximum depth. After reaching the end of a branch, it backs up to the most recent previous choice point, continuing this process. Assuming DFS cuts off search at depth d and a constant branching factor b, it requires only a small amount of memory linear in d. However, DFS suffers from an explosion of time complexity $\Theta(b^d)$ in the worst case, since it needs to search the whole tree. DFS also has the problem of how to determine the maximum search depth d. In general, since the value of d required to solve a problem is not known, d must be decided by some heuristics. If d is too small, DFS does not find any solution. On the other hand, if d is chosen too large, DFS results in a large amount of unnecessary tree expansions.

Best-first search (BFS) [65] is an alternative approach that overcomes some of the disadvantages of DFS. BFS is an algorithm that picks up a promising node among all the nodes currently expanded with guidance of a *heuristic estimation*. BFS usually achieves less tree expansions than DFS, if the heuristic estimation is accurate. However, since BFS needs to keep all the nodes explored in working memory, it suffers from a combinatorial explosion of space complexity.

Depth-First Iterative Deepening and the Transposition Table

Iterative deepening overcomes the problem of determining the maximum depth to explore for DFS. This technique first appeared as a better time control scheme in Slate and Atkin's chess program [78]. *Depth-first iterative deepening (DFID) search* first carries out a depth-first search to depth 1. Then, if DFID cannot find a solution, it starts over and performs a depth-first search with depth 2, then depth 3, and so forth, continuing this process until it finds a solution. At first sight, DFID seems less efficient than directly performing DFS to depth d, because of the extra cost of re-expanding previously explored nodes. However, DFID is usually more efficient than a direct d-*ply* search [1] in practice, by using information from previous searches to achieve a better node ordering. When DFID performs a new iteration, a promising path explored in the previous iteration is usually searched first. The previous iteration gives DFID reliable information on ordering nodes for the next iteration. The cost paid for the previous iteration is a small price to pay for gaining the benefits of searching a promising node first with a larger depth d.

One of the practical enhancements to DFID is the use of a *transposition table*, a large cache that keeps results of previous search efforts. This approach appeared in early chess programs [26, 78]. There are two benefits of incorporating a transposition table into DFID:

1. The search space of many domains is not a tree, but a graph. More than one path can lead to the same node — a so-called *transposition*. If a cached node has been explored deep enough, DFID does not need to explore that node again, thus saving considerable search effort.

2. DFID repeatedly re-visits many nodes. Even if the search effort stored in the transposition table is not sufficient to avoid searching the subtrees, the transposition table information helps DFID to improve node ordering. For example, in game-tree search, the best move from a previous

---

[1] The term *ply* was introduced by Samuel [67]. The ply count represents a distance from the root, and is the same notion as depth.

iteration has a high probability to be the best move for a new iteration, and a considerable amount of work is saved by examining this move first.

2.1.3 Depth-First versus Best-First Search for AND/OR Tree Search

Although both BFS and DFS (and DFID) seem to have merits and limitations, research on search algorithms reveals that some best-first search algorithms can be turned into depth-first search algorithms. Examples in single-agent search are A* and its depth-first version IDA* [42]. In game-tree search, SSS* [79] has a depth-first variant MT-SSS* [61].

Sections 2.1.4 and 2.1.5 review work on two best-first search algorithms for AND/OR trees, AO* and proof number search. Sections 2.1.6 to 2.1.8 deal with the reformulation of these algorithms as depth-first search algorithms, leading to the algorithms PN*, PDS, and df-pn.

2.1.4 The AO* Algorithm

AO* is a representative best-search algorithm for AND/OR trees [48, 59]. However, because AO* is quite complicated and does not achieve high performance, practitioners prefer using the other algorithms explained later.

AO* uses a heuristic function $h(n)$ and a cost function $f(n)$. Assuming that AO* needs to pay a non-negative cost $c(n,m)$ when moving from node n to m, a heuristic function $h(n)$ to estimate the cost required to prove node n is defined by programmers. As the A* algorithm finds an optimal solution (called *admissibility*), similarly AO* guarantees admissibility if $h(n)$ never overestimates the real cost. Let $n_1, \cdots, n_k$ be the children of n. If $h(n)$ is admissible, $f(n)$ is a lower bound of finding a solution for a subtree rooted at n, based on the tree currently explored:

1. For a **true** terminal node, $f(n) = 0$.
2. For a **false** terminal node, $f(n) = \infty$.
3. For a leaf node n, $f(n) = h(n)$.

4. For an interior OR node n,

$$f(n) = \min(f(n_1) + c(n, n_1), \cdots, f(n_k) + c(n, n_k)).$$

5. For an interior AND node n,

$$f(n) = \sum_{i=1}^{k} (f(n_i) + c(n, n_i)).$$

AO* works as a best-first search with guidance of $f(n)$. AO* keeps the whole tree in memory, tracing a most promising path at each step by selecting the child having the minimal $f(n)$ at each OR node. AO* expands a leaf node of a most-promising path, and revises the cost function by propagating cost estimates upward to the root. This process continues until the root is solved.

One problem of AO* is that it needs to keep all explored nodes in working memory. As a result, AO* suffers from combinatorial explosion of the search space.

2.1.5 Proof-Number Search

Allis introduced the notion of proof and disproof numbers for his best-first search algorithm called *proof-number search (PNS)* [4]. The idea of proof numbers originates from McAllester's conspiracy numbers, which measure the reliability of an evaluation in minimax search [49].

The proof number of a node is defined as the minimum number of leaf nodes that must be *proven* to prove that the node is a win for the first player, while the disproof number is the minimum number of leaf nodes that must be *disproven* in order to disprove the node. Proof and disproof numbers can be viewed as an estimate of how easy it is to prove or disprove a tree.

Let n be a node and $n_1 \cdots n_k$ be n's children. Since only one proven child suffices to win at an OR node, while all children must be proven to win an AND node (and vice versa for disproof), the proof number $\mathbf{pn}(n)$ and the disproof number $\mathbf{dn}(n)$ of a node n are defined as follows:

1. For a proven node n, $\mathbf{pn}(n) = 0$ and $\mathbf{dn}(n) = \infty$.

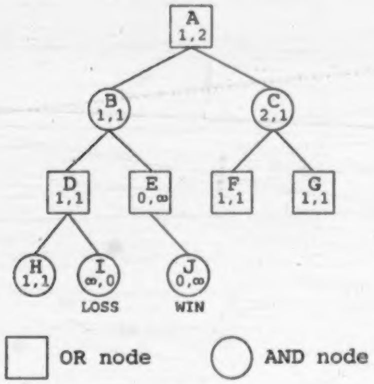


Figure 2.1: Proof and disproof numbers.

2. For a disproven node n, $\mathbf{pn}(n) = \infty$ and $\mathbf{dn}(n) = 0$.

3. For an unknown leaf node n, $\mathbf{pn}(n) = \mathbf{dn}(n) = 1$.

4. For an interior OR node n,

$$\begin{aligned}\mathbf{pn}(n) &= \min(\mathbf{pn}(n_1), \cdots, \mathbf{pn}(n_k)) \\ \mathbf{dn}(n) &= \mathbf{dn}(n_1) + \cdots + \mathbf{dn}(n_k).\end{aligned}$$

5. For an interior AND node n,

$$\begin{aligned}\mathbf{pn}(n) &= \mathbf{pn}(n_1) + \cdots + \mathbf{pn}(n_k) \\ \mathbf{dn}(n) &= \min(\mathbf{dn}(n_1), \cdots, \mathbf{dn}(n_k)).\end{aligned}$$

Figure 2.1 illustrates an example of proof and disproof numbers. The proof number is written on the left inside a node and the disproof number on the right.

Proof numbers can be seen as a special case of the cost function in AO* with $h(n) = 0$ for a proven n, $h(n) = \infty$ for a disproven n, $h(n) = 1$ for each unproven leaf node n, and $c(n, m) = 0$ for all edges. However, PNS has better criteria for selecting a most promising node at both AND and OR nodes.

PNS maintains a proof and a disproof number for each node. The leaf node to expand next is chosen in a best-first manner. Starting from the root,

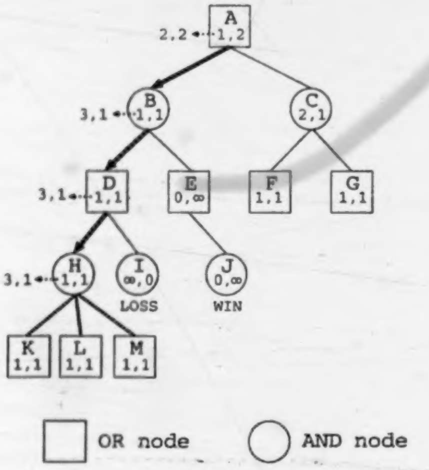


Figure 2.2: Snapshot of expanding a tree in PNS.

PNS traverses the tree by continuously selecting a child whose (dis)proof number is minimum at OR (AND) nodes, until it reaches a leaf node called the *most-promising node.* PNS expands that node and recomputes the proof and disproof numbers on the path to the root. This process continues until the root is either proven or disproven.

Figure 2.2 illustrates an example of tree expansion in PNS. The most-promising node is H. Expanding H yields 3 new nodes, K, L, and M. Next, PNS backs up the updated proof and disproof numbers of the nodes on the path to the root. For example, A's proof and disproof numbers become 2 and 2 respectively.

Allis *et al.* applied PNS to the games Qubic, Go-Moku, Awari, give-away chess [2], and checkmating problems in chess [10]. Qubic and Go-Moku were solved. PNS performed much better than state of the art $\alpha\beta$ algorithms.

2.1.6 The PN* Algorithm

To overcome the large memory requirement of AO*, Seo's PN* converts a special case of AO* into a depth-first search algorithm that requires less memory [77]. Like AO*, PN* explores a promising node first. The equivalence between AO* and PN* is proven by Nagai *et al.* [58].

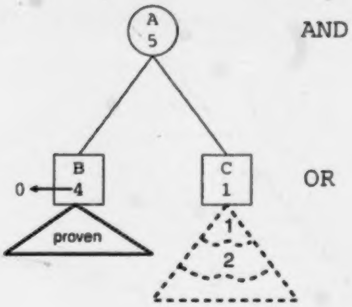


Figure 2.3: Multiple iterative deepening.

PN* employs depth-first iterative deepening. Unlike normal iterative deepening methods, PN* uses the proof number as a threshold, not the search depth. PN* begins searching the root with a threshold of **pn** = 2. If neither a proof nor a disproof is found, PN* increments **pn** and tries to prove the root within the threshold of **pn** = 3. PN* continues this process, until there is a proof of the root or the root is disproven. Since this process re-expands many nodes already visited, PN* uses a transposition table to store previous search efforts, i.e., the proof numbers of visited nodes. Before a node is explored, PN* checks if a proof number for that node is saved in the transposition table. If this is the case, the proof number of that node is initialized to the number saved in the table, otherwise it is set to 1. Similarly, when selecting a promising child of a node to explore, PN* checks the transposition table for information on the children.

PN* performs iterative deepening at all OR nodes, which is called *multiple iterative deepening*. This technique avoids the problem of a gap between the current threshold and the proof number for an AND node, when PN* expands the AND node and one of the children is proven. Figure 2.3 illustrates an example. Currently A's proof number is $\mathbf{pn}(B) + \mathbf{pn}(C) = 4 + 1 = 5$. Assume that A is explored with a threshold of 6, B is selected and proven. The proof number for A then decreases to 1, since C is the only node left to prove. However, A's threshold is still 6, much larger than A's current proof number. Searching C with such a large threshold can result in a large amount of tree exploration. Multiple iterative deepening reduces the search nodes dramatically by assigning smaller initial thresholds to search the children of AND nodes. One might be concerned about the overhead of re-expansion of interior nodes. However, this price is relatively small. Seo implemented PN* for his tsume-shogi (Japanese chess mating problem) solver and showed that the ratio of node re-expansions is about 20 percent experimentally. With further enhancements, Seo's tsume-shogi solver performed far better than previous solvers based on AO* [29].

2.1.7 The PDS Algorithm

Seo's PN* uses only proof numbers. Nagai's PDS (Proof and Disproof-number Search) algorithm enhances PN* by using both proof and disproof numbers [55]. Its behavior is almost the same as PNS, but it uses less memory and explores fewer interior nodes by taking advantage of the techniques invented for PN*.

PDS has two thresholds, one for proof numbers and one for disproof numbers. PDS expands a node n while $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ do not both exceed their thresholds. It uses a transposition table to store proof and disproof numbers for visited nodes, and performs multiple iterative deepening at all nodes. Once both proof and disproof numbers exceed their threshold, the threshold of either the proof or the disproof number is incremented. Let $n.\phi$ and $n.\delta$ be as defined follows:

$$
\begin{aligned}
n.\phi &= \begin{cases} \mathbf{pn}(n) & (n \text{ is an OR node}) \\ \mathbf{dn}(n) & (n \text{ is an AND node}) \end{cases} \\
n.\delta &= \begin{cases} \mathbf{dn}(n) & (n \text{ is an OR node}) \\ \mathbf{pn}(n) & (n \text{ is an AND node}) \end{cases}
\end{aligned}
$$

Then, the incrementing strategy of PDS is: (a) If $n.\phi \leq n.\delta$, increment $n.\phi$; (b) otherwise increment $n.\delta$. Intuitively PDS aims at a proof if a search tree seems to be easier to prove, and at a disproof if it seems easier to disprove. By using both proof and disproof numbers, PDS is good not only at proofs but also at disproofs.

Nagai experimented with PDS on Othello and random game trees and achieved a better performance than PNS, AO*, and PN* [54, 55].

2.1.8 The Df-pn Search Algorithm

Unlike PNS, in PDS a most-promising node does not exist. Later, Nagai invented df-pn (depth-first proof-number) search [56], which turns PNS into a depth-first search algorithm by generalizing the ideas behind PDS. As a depth-first search, df-pn can expand less interior nodes and use a smaller amount of memory than PNS. Like PNS, it always expands a most-promising node.

Figure 2.4, adapted from [56], presents pseudo-code of the df-pn algorithm. The code is written using the ϕ and δ notation introduced in Section 2.1.7. In the pseudo-code, df-pn returns a value of whether the first-player wins or loses.

Df-pn utilizes two thresholds like PDS, one for proof and the other for disproof numbers. While the iterative deepening methods such as PN* and PDS have global thresholds, df-pn's thresholds work as local thresholds at each recursive call. This approach is similar to recursive best-first search [43]. The main function *Df-pn* initializes both thresholds to infinity, and then calls the recursive function *MID* that iterates over nodes. When returning from *MID*, the root node is either proven or disproven. In practice, one more value that represents **unknown** can be returned, such as if df-pn uses up the time required in the tournament. *MID* traverses the subtree below node n in a depth-first manner. It explores nodes while proof or disproof numbers do not exceed the threshold, or until it finds a terminal node that determines a winner. In the code, *IsTerminal* checks if n is a terminal node, while *WinForCurrentNode* checks whether a terminal node is a win or a loss. When a node n is expanded, the best child n_c in terms of proof and disproof numbers is selected by *SelectChild* for a recursive call to *MID* with the following new thresholds: $n_c.\delta$ is set to the minimum of the current threshold for n and the value when n's child with the second smallest δ becomes the most-promising node during the exploration of n_c's subtree. Note that $n.\phi$ corresponds to $n_c.\delta$ because of the duality of proof and disproof numbers. In order to set $n_c.\phi$, the

```
// Set up for the root node
int Df-pn(node $r$) {
  $r.\phi = \infty$;  $r.\delta = \infty$;
  MID($r$);
  if ($r.\delta = \infty$)
    return win_for_root;
  else
    return loss_for_root;
}

// Iterative deepening at each node
void MID(node &$n$) {
  TTlookup($n$,$\phi$,$\delta$);
  if ($n.\phi \le \phi$ || $n.\delta \le \delta$) {
    // Exceed thresholds
    $n.\phi = \phi$; $n.\delta = \delta$;
    return;
  }
  // Terminal node
  if (IsTerminal($n$)) {
    if (WinForCurrentNode($n$)) {
      $n.\phi = 0$; $n.\delta = \infty$;
    } else{
      $n.\phi = \infty$; $n.\delta = 0$;
    }
    TTstore($n$,$n.\phi$,$n.\delta$);
    return;
  }
  GenerateMoves($n$);
  // Store larger proof and disproof
  // numbers to detect repetitions
  TTstore($n$,$n.\phi$,$n.\delta$);
  // Iterative deepening
  while ($n.\phi > \Delta$Min($n$)  &&
          $n.\delta > \Phi$Sum($n$))  {
    $n_c$ = SelectChild($n$,$\phi_c$,$\delta_2$);
    // Update thresholds
    $n_c.\phi = n.\delta + \phi_c$ - $\Phi$Sum($n$);
    $n_c.\delta$ = min($n.\phi$,$\delta_2$ + 1);
    MID($n_c$);
  }
  // Store search results
  $n.\phi = \Delta$Min($n$); $n.\delta = \Phi$Sum($n$);
  TTstore($n$,$n.\phi$,$n.\delta$);
}
```

```
// Select the most promising child
node SelectChild(node $n$, int &$\phi_c$,
                    int &$\delta_2$)  {
  node $n_{best}$;
  $\delta_c = \phi_c = \infty$;
  for (each child $n_{child}$ of $n$) {
    TTlookup($n_{child}$,$\phi$,$\delta$);
    // Store the smallest and second
    // smallest $\delta$ in $\delta_c$ and $\delta_2$
    if ($\delta < \delta_c$) {
      $n_{best} = n_{child}$;
      $\delta_2 = \delta_c$; $\phi_c = \phi$; $\delta_c = \delta$;
    }
    else if ($\delta < \delta_2$)
      $\delta_2 = \delta$;
    if ($\phi = \infty$)
      return $n_{best}$;
  }
  return $n_{best}$;
}

// Compute the smallest $\delta$ of
// $n$'s children
int $\Delta$Min(node $n$) {
  int $min = \infty$;
  for (each child $n_{child}$ of $n$) {
    TTlookup($n_{child}$,$\phi$,$\delta$);
    $min$ = min($min$,$\delta$);
  }
  return $min$;
}

// Compute sum of $\phi$ of $n$'s children
int $\Phi$Sum(node $n$) {
  int $sum = 0$;
  for (each child $n_{child}$ of $n$) {
    TTlookup($n_{child}$,$\phi$,$\delta$);
    $sum = sum + \phi$;
  }
  return $sum$;
}
```

Figure 2.4: Pseudo-code of the df-pn algorithm.

gap between $n.\delta$ and the sum of all ϕ of n's children is computed, and then $n_c.\phi$ is set to the sum of this gap and the current ϕ for n_c.

Because df-pn is an iterative deepening method that expands interior nodes again and again, the heart of the algorithm is the transposition table. *TTstore* stores proof and disproof numbers of a node in the table. *TTlookup* checks the table for information on proof and disproof numbers of a node. If no result is found, both numbers are initialized to 1.

Nagai applied df-pn to tsume-shogi, and df-pn contributed to make his solver the current best solver, by solving all the existing hard problems whose solution sequence is longer than 300 steps [56].

2.1.9 Simulation

Tree *simulation* was invented by Kawano to effectively deal with useless interposing piece drops in tsume-shogi [30]. Later, Tanase extensively applied this idea in his $\alpha\beta$ search engine to reduce the overhead of calling the tsume-shogi solver inside the normal search [80].

Assume that P is a proven node and Q is a "similar" one that simulation wants to prove. Simulation borrows moves from P's proof tree at each OR node to try to find a quick proof of Q. Thus, if simulation returns a proof for Q, Q is proven as well. Otherwise, Q's value is unknown.

Compared to a normal search, simulation requires much less effort to confirm whether a position is proven or not. Even with good move ordering, a newly created search tree is typically much larger than an existing proof tree. Also, since moves are borrowed from the transposition table at OR nodes, there is no need to invoke the move generator there. Figure 2.5 presents pseudo-code for simulation. The notation of Figure 2.4 is used. n_{proof} is a node that has already been proven, and n_{sim} is a similar node to try to prove. *WinForORNode* checks if n is a proven terminal node, while *IsORNode* checks if n is an OR node. *TTstoreProof* saves a proof in the transposition table. *IsProofInTT* checks if n's proof is saved in the transposition table. *RetrieveWinningMove* retrieves the winning move for an OR node n from the transposition table if n is proven. One possible refinement is that n_{proof} can be the transposition table key of n_{proof}, since all the algorithm needs is to retrieve information in the transposition table.

Figure 2.6 illustrates an example of how simulation works. Assume that Figure 2.6(a) is A's proof tree, and A' in Figure 2.6(b) is a similar position to prove. Normal search algorithms generate all moves (i.e., $m1$ and $m6$) at OR node A'. However, simulation generates only $m1$ at A', because A is proven by m_1. Nodes below G are not explored by simulation. Moreover, $m1$ can be saved in A's transposition table entry when A is proven. $m1$ for A' can be retrieved from A's table entry. Only checking the legality of move $m1$ for A' is necessary. At AND node B', simulation generates all moves, $m2$ and $m3$, because all branches must be proven. This process is recursively called at C' and D'. Only $m4$ at C' and $m5$ at D' are tried to check if A''s proof tree can be constructed. The proof tree is confirmed by reaching E' and F' and checking that these positions are proven.

2.1.10 The $\alpha\beta$ Algorithm

The $\alpha\beta$ algorithm [41] is the most popular algorithm used in game-playing programs to determine a next move. In the $\alpha\beta$ framework, the first player tries to maximize his or her advantage, while the second player tries to minimize it. $\alpha\beta$ evaluates a leaf node by calling an evaluation function that approximates the chance of the first player winning. $\alpha\beta$ explores nodes in a depth-first manner to compute the best score at the root, based on the *minimax* framework. In this framework, the score at each node is calculated from the leaf nodes in a bottom-up manner. The first player maximizes the score of the children at each node, while the second player minimizes the scores from the children.

Although $\alpha\beta$ returns a numeric score, it can be applied for AND/OR trees, for example, by assigning a score of 1 for a proof, 0 for an unknown value, and -1 for a disproof.

$\alpha\beta$ utilizes a *search window* defined by two bounds, α and β, which represent lower and upper bounds on the score of a tree. The search window is narrowed during search, and used for pruning subtrees if the score of a node is proven to be outside of the window. Many variants and enhancements have

```
int Simulation(node $n_{proof}$, node $n_{sim}$) {
  // Terminal node
  if (IsTerminal($n_{sim}$)) {
    if (WinForORNode($n_{sim}$)) {
      TTstoreProof($n_{sim}$);
      return true;
    }
    return unknown;
  }
  // Check if a proof is saved in the transposition table
  if (IsProofInTT($n_{sim}$))
    return true;
  // Try to construct $n_{sim}$'s proof tree
  if (IsORNode($n_{sim}$)) {
      move = RetrieveWinningMove($n_{proof}$);
      // Try only one move
      if (move == NO_MOVE || !IsLegal($n_{sim}$, move))
        return unknown;
      $child_{proof}$ = MakeMove($n_{proof}$, move);
      $child_{sim}$ = MakeMove($n_{sim}$, move);
      result = Simulation($child_{proof}$,$child_{sim}$);
      if (result == true)
        TTstoreProof($n_{sim}$);
      return result;
  } else {
    GenerateMoves($n_{sim}$);
    // Check if all moves lead to true
    for (each move $m$ of $n$) {
      $child_{proof}$ = MakeMove($n_{proof}$, m);
      $child_{sim}$ = MakeMove($n_{sim}$, m);
      if (Simulation($child_{proof}$,$child_{sim}$) == unknown)
          return unknown;
    }
    TTstoreProof($n_{sim}$);
    return true;
  }
}
```

Figure 2.5: Pseudo-code of simulation.

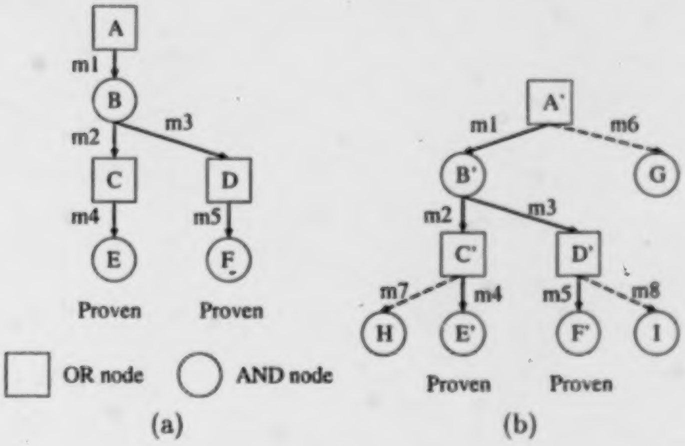


Figure 2.6: Kawano's simulation.

been developed over the years [47, 64, 70], but a transposition table is almost always used.

2.1.11 Replacement and Garbage Collection Schemes

When search algorithms use up all transposition table entries, some table entries must be deleted to save new search results. Replacement schemes overwrite a transposition table entry, when a collision occurs. On the other hand, garbage collection schemes delete a large number of transposition table entries at a time when the transposition table becomes full or almost full. Breuker investigated effective methods for replacement schemes [11]. The most effective scheme computes the number of nodes of a subtree. The node that contains a larger subtree is preserved in the transposition table. Nagai investigated garbage collection schemes [55]. *SmallTreeGC*, which he later implemented for his tsume-shogi solver [56], is an effective method. In SmallTreeGC, each table entry contains the number of nodes of its subtree. When SmallTreeGC is invoked, it keeps track of all table entries and discards the entries that contain small subtrees until a certain amount of table entries are deleted. By incorporating this technique, Nagai's tsume-shogi solver was able to solve hard problems with a much smaller amount of memory than other tsume-shogi solvers.

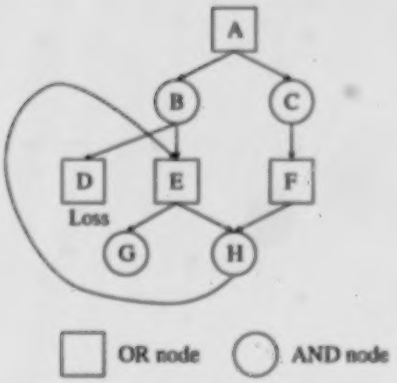


Figure 2.7: The GHI problem.

2.1.12 Summary

Over the last 20 years, many improvements have been developed for AND/OR tree search algorithms. In particular, the notion of proof and disproof numbers was shown to be very effective. However, how to best adapt the algorithms surveyed in this section to harder search domains such as Go is still an open question. Moreover, if the search space is a directed cyclic graph and transposition tables are used, current algorithms may return incorrect results. The next section deals with this problem.

2.2 The GHI Problem

The last section explained that enhanced AND/OR tree search algorithms make use of transposition tables to save search effort. However, if the search space includes cycles, cached results may be flawed because they ignore the path used to reach the position. This is the so-called GHI problem [60]. This section explains the problem and surveys previous approaches to solve it.

2.2.1 Problem Description

With the help of Figure 2.7 the GHI problem for AND/OR trees is explained. There are two scenarios in which the GHI problem can occur, depending on the rules of the game.

In the first scenario, which is called *first-player-loss*, a repetition is considered to be a loss for the *first player*, the player to play at the root node. Examples are checkmating problems in chess and shogi (tsume-shogi), since a repetition does not help the first player who is trying to checkmate. Assume D in the figure is a loss for the first player, and this result is stored in the transposition table. Let G be a win for the first player. Then a search starting from A in the following order leads to the wrong result:

1. Search $A \rightarrow B \rightarrow E \rightarrow H \rightarrow E$. A loss is stored in the table entry for H, because the position repetition cannot be avoided.
2. Search $A \rightarrow B \rightarrow D$. A loss is stored for AND node B.
3. Expand $A \rightarrow C \rightarrow F \rightarrow H$. A table look-up for H retrieves a loss which is backed up to F and C.
4. A is now incorrectly labeled as a loss because losses are stored for both successors B and C. However, A is a win by the sequence $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow G$.

In the first-player-loss scenario, the GHI problem only causes invalid disproofs (first-player losses). Programs can avoid the GHI problem, accepting a loss of performance, by not storing any disproofs caused by repetitions.

The other scenario for GHI, which is called *current-player-loss*, occurs when a repetition is declared to be a loss for the player who repeats the position. For instance, the situational super-ko (SSK) rule in Go declares that any move that repeats a previous board position is illegal. In this scenario, using a transposition table can lead to errors in both ways: it can change a loss into a win or a win into a loss. For example, in Figure 2.7, now assume that G is a loss for the player to move at the root:

1. Search $A \rightarrow B \rightarrow E \rightarrow H$. H is stored as a win because the opponent does not have a legal move at H.
2. Search $A \rightarrow C \rightarrow F \rightarrow H$. The win stored for H is backed up and a win is stored for C as well.

3. A is now incorrectly labeled as a win since C's table entry shows a win. However, A is a losing position, since the sequences $A \rightarrow B \rightarrow D$, $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow G$ and $A \rightarrow C \rightarrow F \rightarrow H \rightarrow E \rightarrow H$ all lose.

This scenario does not occur in checkmating problems where only one player's king is under attack. However, van der Werf *et al.* point out that when using the SSK rule in Go, this scenario can lead to invalid proofs [83]. In their work the problem is avoided by storing a separate hash entry for each path leading to a node. Unfortunately, this resulted in over 1,000 times larger searches when solving Go on a 4 × 4 board.

Avoiding the GHI problem is crucial, especially when one wants to declare that games are solved by programs. Since even a single flawed transposition table entry can lead to a completely wrong solution, correct techniques must be devised.

2.2.2 Palay's Suggestions

Palay first pointed out the GHI problem and suggested two solutions [60]. The first solution is to refrain from using transpositions. Van der Werf used this approach to solve 4 × 4 Go with the SSK rule [83]. The drawbacks are a large number of expansions of duplicated nodes and large space requirements. Palay's second solution is to continue using a graph representation but attempt to recognize the GHI problem. When GHI is recognized, his solution checks a path from a node which has more than one parent to a node that causes a repetition. The nodes that are on such a path are duplicated to be able to store different results. However, Palay did not implement this strategy, since GHI did not occur so frequently in his tests. He conjectured that the second solution would take additional time since the graph must be revised occasionally.

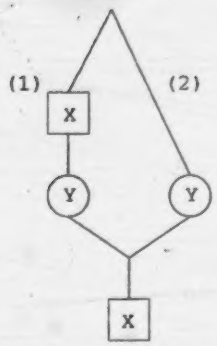


Figure 2.8: Prototypical case of GHI.

2.2.3 Campbell's Analysis

Campbell partially solved the GHI problem for $\alpha\beta$ search [13]. In his algorithm, each transposition table entry contains a field that stores the depth searched below a node. If a transposition is recognized and the depth stored in the table entry is at least as deep as the depth that must be explored, the table information is retrieved and no further search for that node is performed. Campbell classified the GHI problem into two cases, ***draw-first*** and ***draw-last***. These names come from the fact that path $X \rightarrow Y \rightarrow X$ is treated as a draw by repetition in computer chess. Figure 2.8, adapted from [13], illustrates an example. In the ***draw-first*** case, path (1) is explored first and a score is stored in the table entry for X. Then if Y is searched via path (2), Y might be incorrectly computed because of the table entry for X. In the ***draw-last*** case, (2) is explored first and Y's incorrect score is used when reaching Y via (1). This happens because of the implementation of the transposition table. When reaching X via (1), the search depth for X via (2) is shallower than via (1). Therefore, X via (1) must be explored. On the other hand, if Y is reached via (2) first, Y is already explored deeply enough via (1) to reuse the table information on Y.

Campbell noted that draw-first GHI is curable by not storing scores that might cause the GHI problem, while draw-last is incurable. However, Campbell mentioned that in practice most GHI problems can be avoided by combining the $\alpha\beta$ algorithm with iterative deepening [13]. Breuker conjectured that the GHI problem appears much less frequently in iterative deepening $\alpha\beta$ search than in best-first search [12]. These conjectures will be discussed in Chapter 3.

2.2.4 Breuker's Base-Twin Algorithm

Breuker *et al.* proposed the *base-twin algorithm (BTA)* for solving the GHI problem in proof-number search [12]. BTA is described for a 3-valued evaluation model with values *win*, *loss*, and *draw*. If a draw is considered a disproof as in their experiments, this model is the same as the first-player-loss scenario.

BTA uses a *possible-draw* mark combined with the depth of a node to recognize repetitions. To find out which level of the node causes repetitions, BTA utilizes two kinds of nodes: a *base* node to be explored and *twin* nodes that have different parents, and link to their base node, but are not explored. When more than one path reaches identical positions, these positions are not represented by a single node, but split into one base node and one or more twin nodes, which can have different values (i.e. possible-draw marks) than the base node. Whenever a most-proving node is selected, BTA also checks if possible-draw marks can be stored by recognizing repetitions. Possible-draw marks are passed back to parents and when the root of the subtree that causes repetitions is detected, then a real draw is stored in that root. See [12] for details of the algorithm.

Although Breuker *et al.* claim that BTA is a general solution to the GHI problem for best-first search, there are three issues that must be addressed:

1. Since BTA was implemented for a best-first search algorithm that keeps an explicit graph in memory, it is an open question whether BTA is applicable to depth-first search algorithms with limited memory. They concluded in [12] "What remains is solving the GHI problem for depth-first search. This will need a different approach, storing additional information in transposition tables rather than in the search tree/graph in memory. However, Campbell already noted that in depth-first search

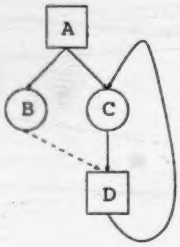


Figure 2.9: An example where BTA fails with the current-player-loss scenario.

the frequency of GHI problems is considerably smaller than in best-first search [13]. The solution of the GHI problem for depth-first search remains a nearly theoretical exercise." However, curing the GHI problem for depth-first search algorithms using proof and disproof numbers is necessary, because these algorithms combine properties of best-first search and depth-first search, and GHI occurs more frequently than in iterative deepening $\alpha\beta$.

2. The cycle detection scheme in BTA does not work with the current-player-loss scenario. Figure 2.9 illustrates an example. Assume that B has not been expanded, and it is currently unknown whether B has child D. Then assume that BTA explores $A \rightarrow C \rightarrow D \rightarrow C$. Since this repetition occurs at C, a score (disproof in this case) is saved in C's transposition table entry without any condition. Then, assume that path $A \rightarrow B \rightarrow D$ is searched, and recognizes D as a child of B. Next, BTA reaches C by expanding D. In BTA, C's entry has a disproof and a disproof is retrieved from the entry. However, this is incorrect, since the *second* player cannot make a move at C if it is reached via $A \rightarrow B \rightarrow D \rightarrow C$.

3. All the possible-draw marks are removed for each iteration of proof-number search. The deletion of possible-draw marks is necessary in BTA since it is path-dependent information. Figure 2.10 illustrates an example which returns an incorrect draw score when possible-draw marks are not cleared. Assume that the first-player-loss scenario is used in

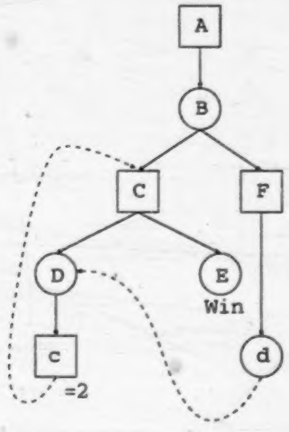


Figure 2.10: An example showing why BTA must remove possible-draw marks.

> Figure 2.10. If path $A \to B \to C \to D \to c$ is followed, a possible-draw mark is stored at c. Since BTA keeps C's depth combined with the possible-draw mark, the depth of 2 is stored in c to indicate that C is the node involving a repetition. Then, if the possible-draw mark at c is not deleted and c is reached via $A \to B \to F \to d \Rightarrow D \to c$, the mark at c is passed back to d (and F). As a result, a real draw is stored at F. However, this is incorrect, since F is a win via path $F \to d \Rightarrow D \to c \Rightarrow C \to E$.
>
> Hence, as long as real draws are not stored, the nodes causing repetitions must be explored again and again to mark possible-draws, resulting in much tree expansion overhead.

2.2.5 Nagai's Approach

Nagai proposed a solution to the GHI problem for df-pn [56]. He applied this modified df-pn to tsume-shogi problems, a first-player-loss scenario. In his algorithm, df-pn first sets large thresholds of proof and disproof numbers at the root. These thresholds are not ∞ as in the original df-pn, but $\infty - 1$. In this model, Nagai assumes that $\infty - 1 < \infty$, since a large integer is used to express ∞ in practice. In case of a repetition, df-pn simply returns to the parent node without storing a disproof. If a proof for the root is found, the proof tree is guaranteed to be repetition-free. However, if df-pn returns to the root by exceeding one of the large thresholds $\infty - 1$ for proof and disproof numbers, df-pn re-searches with a threshold of ∞. If the root position is reached by a move, this move is now considered disproven. A similar process is performed at all interior nodes. If df-pn exceeds one of the $\infty - 1$ thresholds, it re-searches with a threshold of ∞, assuming that that node is a disproven position.

There are two drawbacks of Nagai's approach:

1. It may take a long time for the proof or disproof numbers to exceed the preset threshold. For example, if there are many branches, the thresholds for expanding a node are much smaller than $\infty - 1$, since df-pn locally sets the thresholds based on the proof and disproof numbers of the children. Hence, because this approach has to wait for proof or disproof numbers of all the children to reach $\infty - 1$, it is impractical for detecting disproofs with repetitions. Nagai measured the ability of his tsume-shogi solver only with positions that can be proven. He did not measure the overhead incurred by this approach, or the performance in positions where a node must be disproven.

2. Nagai's approach also does not work with the current-player-loss scenario. Since this approach does not use any path information, it cannot store two different path-dependent results for one node. Again, Figure 2.9 serves as an example. In this figure, D via $A \to B \to D$ is a proven node, since $A \to B \to D \to C \to D$ is not allowed. On the other hand, D via $A \to C \to D$ is a disproven node, since $A \to C \to D \to C$ is illegal. D's value cannot be determined without considering the path used.

2.2.6 Other Related Work

According to [12], Thompson noticed that his tactical chess analyzer suffered from the GHI problem. He cured it by using a DCG (directed cyclic graph) representation. When a node was expanded, his analyzer took the history into account to avoid returning an incorrect result. However, when leaf nodes were evaluated, the history was not considered, possibly resulting in incorrect evaluation values.

Baum and Smith suggested a solution to the GHI problem for their best-first search algorithm [5]. Their algorithm stores the whole DCG in memory to be able to check all ancestors and descendants of a node. Then, if node P spawns a child Q and Q has another parent P', their method checks if the ancestors of P and the descendants of Q contain P'. If this is the case, Q is split into two nodes to be able to store different results. However, this idea was not implemented. The authors conjectured that a low storage algorithm would probably be too costly.

Schijf *et al.* investigated proof-number search in domains where the search graphs are DAGs (directed acyclic graphs) and DCGs [75]. They observe that in practice it is not necessary to compute proof and disproof numbers correctly for DAGs, as long as correct results are returned. Three algorithms for DCGs are presented:

1. The *tree method* does not use transpositions, which has the disadvantage of not reusing results, while correctness is always guaranteed.

2. In the *DAG method*, two classes of moves are defined: *conversion* moves are irreversible and *non-conversion* moves may be reversible. The DAG method maps identical positions to a single node for conversion moves, while identical positions reached by non-conversion moves are treated as different nodes. This approach is also applied in the solution of 5 × 5 Go with Japanese and Chinese rules by van der Werf *et al.* [83]. This approach can cure the GHI problem, but a disadvantage is that duplicated searches are performed for all nodes with non-conversion moves.

3. The *DCG method* maps identical nodes to a single node unless a cycle is created. If a repetition is detected, a node creating a cycle is mapped to a second node and recognized as a disproven leaf node. Identical positions

are mapped to at most two nodes. Although the DCG method is shown to be effective in their experiments in chess, as pointed out in [75] this approach sometimes results in wrong disproofs.

2.2.7 Summary

More than 20 years have passed since Palay pointed out the GHI problem. Although many solutions were presented, programs still suffer from the GHI problem. Algorithms are either less efficient to guarantee correctness, or incorrect in order to not degrade the performance.

2.3 Previous Research on Computer Go

This section deals with relevant work on computer Go. Research related to tsume-Go is mainly addressed. Good overviews of computer Go in general are available in [9] and [53].

2.3.1 Tsume-Go Solvers

GoTools

Wolf's *GoTools* has been the best tsume-Go solver for 15 years [86, 88]. GoTools specializes in solving completely enclosed positions. GoTools uses a depth-first search algorithm. Unlike most game-playing programs, GoTools does not perform iterative deepening, but traverses a tree as deeply as possible until reaching a terminal node. This search strategy is similar to SAT (Satisfiability) solvers [21] such as SATZ [46]. If a node is either proven or disproven, the result for that node is stored in a transposition table.

GoTools contains an evaluation function that includes look-ahead aspects, powerful rules for static life and death recognition, and learning of dynamic move ordering from the search [90]. The latest version of GoTools has three kinds of parameters, tuned by genetic algorithms [63]: *Static* weights (46 parameters) are mainly for the static evaluation of positions. *Dynamic* weights (10 parameters) are used to order moves based on the results of tree searches performed. *Pruning* weights (14 parameters) give criteria for forward pruning that sacrifices correctness of solutions in order to speed up the search. GoTools has five modes for searches: One mode always ensures correctness of the answers: the other four heuristic search modes use different degrees of forward pruning [90].

One of the most important enhancements in GoTools is to order moves from the subtrees explored [90], which has similarities with the *killer heuristic* [1] and the *history heuristic* [70]. Assume that the first player makes a move m_1 at position P, and the second player refutes m_1 by playing m_2. Then, the first player next tries move m_2 at P, since this is a killer move that reduces the chance of the second player to win. The moves refuting opponent moves at subsequent positions also get some credit to achieve better move ordering.

One research question is to compare GoTools with a tsume-Go solver in which techniques such as proof and disproof numbers are implemented. For example, one possible advantage of the df-pn algorithm is that it uses the transposition table more extensively. Only solved positions are saved in the transposition table in GoTools, while in df-pn proof and disproof numbers of previous iterations are stored in the transposition table to improve the order of tree expansion [56].

Eye Databases

Since making two eyes is the most common way to prove that stones are alive, an early detection method for whether eyes can be created or not can result in a great reduction of the search depth. Dyer created eye databases [22] and Yamashita uses a similar approach in the tsume-Go solver in his Go program *Aya* [91]. Cazenave created a database that enumerates a large number of possible patterns of eyes up to size 5 × 3 [16]. Large databases were successful in many domains such as chess [81], checkers [44], and the 15-puzzle [20]. There is, however, one important difference between Cazenave's work and databases for other domains. For the other databases, once programs find a pattern in the databases that matches a position, it is always a perfect answer without any conditions. In contrast, whether blocks in Go are alive or dead can depend on conditions outside of the pattern. Cazenave's database contains additional

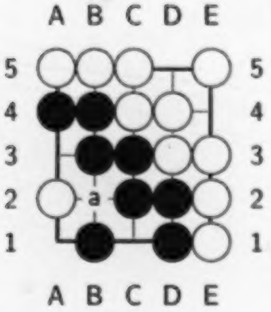


Figure 2.11: An example of safe stones but an unsafe territory.

information on external conditions that must be satisfied to match patterns, such as the number of outside liberties*.

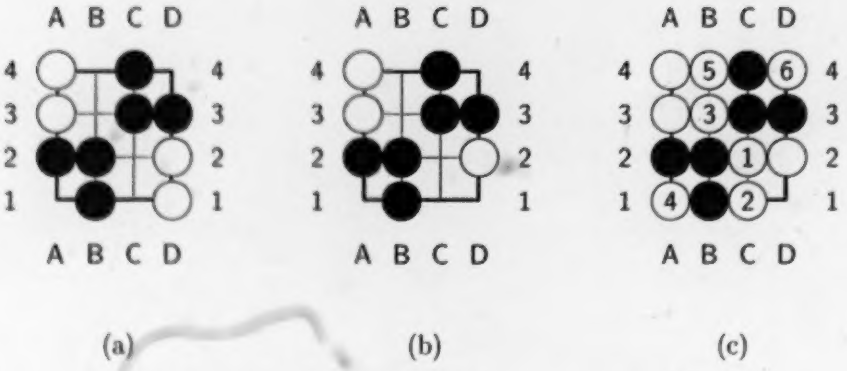
As a result, Cazenave's database requires 32 bits of storage per pattern, while the other databases only need 1 or 2 bits (e.g., to store a win, loss or draw). The size of the database is greatly reduced by making use of domination between positions and by storing only smaller patterns. Cazenave incorporated the database with external conditions into his tsume-Go engine based on proof-number search and achieved better performance.

Static Eye Detection

Vilà and Cazenave presented a static approach to detect large eye shapes [84]. Such eye shapes guarantee life by either dividing it into two eyes or living in seki. They succeeded in statically classifying many eye shapes containing up to 7 points.

2.3.2 Proving Territories Safe

Let a *territory** be an area which is surrounded and controlled by a player. Dead opponent stones can be contained in a territory. Proving territories safe is as essential a task as proving stones safe in Go. Safe territories are very similar and closely related to safe stones. However, there are some cases in which stones are safe while territories are unsafe. Figure 2.11 from [51] shows



(a) (b) (c)

Figure 2.12: An example of Benson's unconditional safety.

an example. In this figure, all black stones are safe. However, the territory surrounded by black stones is not safe, since the white stone inside the territory can live in seki if White plays *a*. Another case is when a surrounded area is so large that the opponent can live inside the area.

Moves committing suicide are not allowed in the methods explained here. Japanese rules forbid suicide. Let the *defender* be the player trying to prove safety, and the *attacker* be the opponent. Benson presented a method for static evaluation of *unconditional safety* of blocks [6]. In Benson's definition, defender blocks are unconditionally safe if the blocks are still safe even after an unlimited number of moves in a row played by the attacker. For example, all the black blocks in Figure 2.12(a) are unconditionally safe. On the other hand, the black blocks in Figure 2.12(b) are not unconditionally safe, since they can be captured by White's consecutive plays (see Figure 2.12(c)).

Popma and Allis generalized the notion of unconditional safety to X *life* [62]. The definition of X life is that the blocks of the defender are still alive after X consecutive passes by the defender, but can be killed after $X + 1$ consecutive passes.

The methods mentioned above are limited in practice, since no defensive moves for threatening moves are allowed. Müller presented a practically more effective method to find the safety of blocks and territories by *locally alternating play* [51] . Locally alternating play allows the players to play moves in

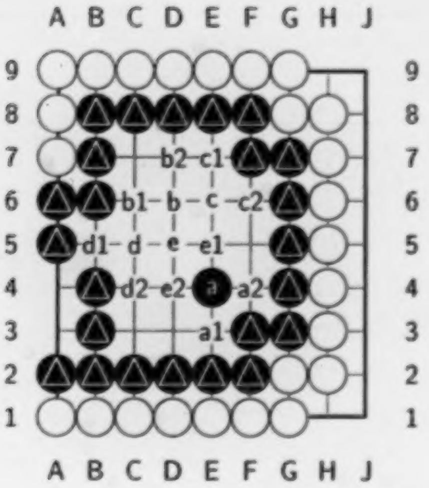


Figure 2.13: An example of safety by locally alternating play.

turns only in one area enclosed by the defender's blocks, starting with the attacker. Figure 2.13 illustrates an example that works effectively with Müller's method. In this figure, the most important technique is the miai* strategy, although there are some other rules to recognize safety. First, one liberty is ensured in an enclosed area at **A4** and **A3**. The *miai* strategy finds two ways for connections to interior points. For example, White cannot prevent Black from connecting the block marked by triangles to *a*, by playing either at *a*1 or at *a*2. Similarly, the black block has connections to *b*, *c*, and *d*. *e* is connected to the block by applying the miai strategy from *a*. As a result, the strategy guarantees a second liberty in the area. Hence, the block and territory are proven to be safe.

2.3.3 Heuristic Estimations of Eyes

Strong computer Go programs have methods to estimate the life and death of blocks. One popular method is to analyze life and death with a combination of exact and heuristic rules.

Ken Chen's *Go Intellect* and Zhixing Chen's *HandTalk* statically analyze the life and death status of blocks [18]. They have evaluation functions for estimating the number of eye points. Fotland's *Many Faces of Go* first classifies the eye shape, then determines a score for eye points by performing tactical searches around the eye spaces [25]. An advantage is that this approach approximates the life and death status faster than a search-based approach. However, one disadvantage is that this method occasionally returns an incorrect answer, which might lose a game. If an efficient and precise method to detect life and death is established, replacing those estimations by a high-performance solver that employs searches will be possible.

2.3.4 Other Related Work

Threat-Based Search

When one player threatens to win immediately, a set of moves that are worth trying can be defined, resulting in a reduced branching factor of search trees. Such an approach, called *threat space search* [3], was used by Allis in combination with PNS to solve Go-Moku [2]. In Go, programs usually have search engines to capture stones in ladders. λ-search was a generalized search algorithm to capture stones [82]. Later, Cazenave presented Generalized Threat Search (GTS) for Atari-Go [17]. In order to find threats (moves that the player must play to win or not to lose), GTS performs searches to a certain depth by allowing a number of consecutive moves played by the same player. Performing these searches is relatively cheaper than doing direct brute-force searches, which improves the performance.

It is still an open question whether GTS is applicable to harder domains, such as tsume-Go, because of the higher complexity of finding a set of threats.

Decomposition Search

Since the larger branching factor in Go makes global search expensive in practice, one idea is to take a divide-and-conquer approach which divides a problem into subproblems that have fewer moves, then solves them and combines their results for the original problem. Müller's decomposition search takes this approach for Go endgames [52]. In decomposition search, a position is divided into subpositions, separated by safe stones. Local searches, based on combinatorial game theory [7], are then performed for each subposition, and allow optimal play using the combinatorial game values of the subproblems. Using this approach, programs can solve a much larger class of endgame problems than with classical minimax-based solvers. However, in basic decomposition search a problem is split into subproblems only at the root node of a search. There are no further splits during the search, so the method fails for example when there is just a single large undivided area in the beginning. In contrast, in one-eye or tsume-Go problems, decompositions can profitably be done inside the search tree. There is usually just one large undivided space in the beginning.

2.4 Research Issues

Although a large amount of resources has been invested in researching search algorithms, the GHI problem, and computer Go, there are still research issues that must be addressed:

- Programmers want to have algorithms that are guaranteed to return correct results without degrading performance. Because of the existence of the GHI problem practitioners are not satisfied with the currently available algorithms. In particular, the only solution for the current-player-loss scenario is to give up all transpositions.
- Programmers are eager to have efficient search methods. This raises the following questions:
    - Nagai's df-pn algorithm has been so successful in tsume-shogi. The performance of his tsume-shogi solver far surpasses that of human players. On the other hand, such a smart search algorithm has not been adapted to the game of Go yet. It is a challenging task to research the efficiency of df-pn for the tsume-Go and one-eye problems.
    - Although decomposition search was shown to be powerful, it is limited to domains where a position is already decomposed at the root

node. The application of decomposition search is therefore currently limited to Go endgames. On the other hand, in applications such as tsume-Go and the one-eye problem, it seems necessary to do decomposition *dynamically* within the search tree. It is still an open question if splitting a position dynamically is feasible.

The thesis will try to resolve these issues in the remaining chapters.

Chapter 3

A General Solution to the GHI Problem

This chapter presents a complete solution to the GHI problem, which has very small overhead for both the first-player-loss and current-player-loss scenarios. This scheme is proven to always return correct answers. Since the idea does not depend on any algorithm-specific features, it can be applied to different game-tree search algorithms. The proposed GHI solution has been implemented for both the df-pn algorithm [56] and $\alpha\beta$ [41]. Experimental results with both algorithms in the domains of Go and checkers show only a very small overhead compared to programs that ignore the GHI problem.

3.1 The Algorithm to Solve the GHI Problem

3.1.1 Overview of the Solution

The outline of the solution to the GHI problem is as follows: When a proven or disproven position stored in the transposition table is reached via a new path, instead of blindly retrieving the result, a search is performed to verify it. If the proof/disproof verifies, the result can be safely reused; otherwise the transposition table entry is treated as a different position. Kawano's simulation [30] is used to reduce the search overhead. For efficiency, this approach requires a good scheme for storing and comparing paths and a technique for minimizing the number of simulation calls.

3.1.2 Duplicating Transposition Table Entries

To reuse the results of previous search efforts, unproven identical positions reached via different paths are considered to be transpositions. The values stored in the transposition table are reused: proof and disproof numbers for df-pn, and minimax values for $\alpha\beta$. When position A is proven via path p, the transposition table entry for A is split into a *base* and a first *twin* table entry. A proof is stored in the twin table entry to indicate that A is proven when reaching A via p. If A is proven via a different path q, another twin table entry for q is created and the new proof is stored there. When reaching A via a path other than p, the proofs of the twin table entries are simulated (see Section 3.1.4). If at least one verifies then that proof is used; otherwise the information from the unproven base table entry is used in the search. Disproofs are handled in the same way.

3.1.3 Encoding Paths

Identical positions reached via different paths can be differentiated by computing a signature of a path. A variant of the Zobrist function, which is used to hash a position into its corresponding transposition table key [92], can be used to encode a path. In the implementation, each transposition table entry contains an additional 64-bit field to encode a signature of the path from the root to a position. Let *MaxMove* be the number of different moves in a game, and *MaxDepth* be the maximum search depth. A precomputed 2-dimensional table R with *MaxMove*×*MaxDepth* random 64 bit integers as entries is prepared to encode a path. The sequence of moves to reach that position is encoded by a technique inspired by Zobrist's method. Let the path p be $(m_1, m_2, \cdots, m_k)$, where m_i are moves. Then p is encoded as follows:

$$\text{code}(p) = R[m_1][1] \oplus R[m_2][2] \oplus \cdots \oplus R[m_k][k]$$

An important property of this path-encoding scheme is that the order of moves is not commutative, since the random table entries for the same move played at different depths are different. For example, the codes of the two paths $p_1 = (m_1, m_2, m_3)$ and $p_2 = (m_3, m_2, m_1)$ are not the same, since code(p_1) = $R[m_1][1] \oplus R[m_2][2] \oplus R[m_3][3]$ is different from code(p_2) = $R[m_3][1] \oplus R[m_2][2] \oplus R[m_1][3]$.

The size of the random table is small enough for current hardware. For example, in the experiments on 19 × 19 Go, setting *MaxMove* = 362 and *MaxDepth* = 50 the size is about 140KB. In games with a large number of different possible moves, such as Shogi or Amazons, a move can be split into two or three partial moves, for example by separating the from-square information from the to-square information. This way *MaxMove* can be greatly reduced, while *MaxDepth* increases by a factor of 2 or 3.

3.1.4 Invoking Simulation for Correctness

Kawano's tree simulation [30] only needs a minimal amount of effort to confirm if a position is proven. Similarly the dual notion of *dual simulation* can be defined to check if a position is disproven.

Since a position identical to a proven or disproven position but reached via a different path is not considered to be a transposition, simulation and dual simulation are utilized to quickly check proofs or disproofs of such positions. Assume that position A was proven via path p. If A is reached via a different path q, simulation can check if A via q can be proven quickly. A proof is borrowed from the twin table entry with path p. If a proof for A via q is verified, an additional twin table entry for the proof of A via q is created. If more than one twin table entry is available, they are tried one after another. However, since proof trees often have the same shape, it is rare that more than one tree simulation is needed. The analogous verification by dual simulation is tried to find disproofs.

3.1.5 Reducing Simulation Calls

Since simulation incurs an overhead to assess the correctness of a transposition table entry, a method to reduce the number of simulation calls is devised. If a node is (dis)proven without detecting a repetition, that node can always be used as a transposition, since it is independent of the path taken by the search.

In this case, the (dis)proof is stored directly in the base table entry, without creating a twin node. If another path leads to that position, this (dis)proof can be reused safely.

3.1.6 Algorithm-Specific Implementation Details

Implementation of the Transposition Table

As in Nagai's work [55], the implementation of the transposition table uses chaining to avoid collisions. The two-level transposition table [11] popularly used for $\alpha\beta$ often loses valuable information from the transposition table, degrading the ability of a solver. However, one implementation problem has to be resolved to make the transposition table work efficiently (see Section 3.4).

Implementation of Df-pn

The following modifications were made to the original df-pn algorithm:

- Proof and disproof numbers in a base table entry are re-initialized to 1 whenever a (dis)proof is saved in a twin table entry. This is because df-pn tends to create large proof and disproof numbers before a (dis)proof is found, which made df-pn unable to solve some positions.

- As in Nagai's GHI solution [56], The thresholds of proof and disproof numbers at the root are initialized to $\infty - 1$, not ∞ as in the original df-pn algorithm. $\infty - 1 < \infty$ holds in the algorithm as in Nagai's method. This is necessary to avoid the GHI problem at the root, since df-pn saves thresholds in the transposition table before expanding a node. If df-pn with this modification returns a proof number of 0 and a disproof number of ∞, or vice versa, it is a correct (dis)proof. Otherwise, df-pn returns the value *unknown*.

Implementation of $\alpha\beta$

The following modifications were made:

- The scheme for transposition table lookups was modified. A normal transposition table entry contains a field that stores the depth searched below a node. If a transposition is recognized, the depth stored in the table entry is at least as deep as the depth that must be explored, and the table entry has a tight $\alpha\beta$ bound that causes a cut-off, then the table information is retrieved and no further search for that node is performed. This strategy is used only for unproven nodes. (Dis)proofs saved in the transposition table are always retrieved without checking the explored depth, since they are correct. This modification not only makes more use of the transposition table but also solves Campbell's draw-last case. The proofs of the theorems that guarantee correctness are given in the next section.

- The current $\alpha\beta$ search implementation uses only the three values (*win*, *unknown*, or *loss*). However, the GHI solution works for the general case of more values in between *win* and *loss*. In the experiments in checkers, a draw is considered as a loss for the first player. To prove a draw, a second search must be performed in which a draw is regarded as a win for the first player. Determining a draw with a single search is not a trivial problem for the $\alpha\beta$ algorithm, since the values *draw* and *unknown* are incomparable. A correct heuristic value can be obtained by performing a sequence of null window searches as in MTD(f) [61] and modifying the transposition table.

3.2 Correctness of the Solution

3.2.1 When Proofs and Disproofs Fit in Memory

Assume that all proven and disproven nodes are stored in the transposition table. The following theorems guarantee correctness of the GHI solution:

Theorem 3.2.1 *The GHI solution does not suffer from the draw-first case.*

Proof. This theorem is proven with the help of Figure 3.1. Although X is an OR node and Y is X's child in Figure 3.1, the only assumption is that

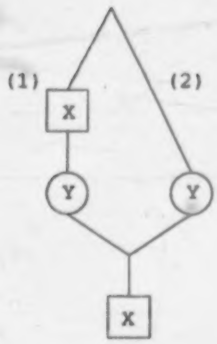


Figure 3.1: Figure used to prove that the GHI solution is free from the draw-first and draw-last cases.

X is Y's ancestor and also Y's descendant. In the draw-first scenario, if proving Y via (1) involves repetitions related to X, X and Y are stored in the transposition table with "via path (1)." Hence, if Y is reached via (2), a search is performed below Y and X in the algorithm. The case of disproofs is similar. Thus the draw-first scenario does not happen in the GHI solution.

■

Theorem 3.2.2 *The GHI solution does not suffer from the draw-last case.*

Proof. In Figure 3.1, with the same assumption as in Theorem 3.2.1, assume that path (2) is explored and X's proof is saved in the transposition table. The following cases have to be considered:

1. *If X via (2) is proven without repetitions, Y is not included in X's proof tree. When X is reached via (1), a proof is immediately retrieved from X's table entry and this is a correct proof, since it does not contain any repetitions. Y is never explored via (1), which would cause the draw-last case.*
2. *If X via (2) is proven with repetitions:*
    (a) If Y via (2) is proven without repetitions, X's proof tree must not be a part of Y's proof tree. When reaching X via (1), X's

proof via (2) is not retrieved because it is stored in the twin table entry via (2). Then, when reaching Y via (1), Y's table entry is retrieved. Because Y's proof tree does not contain any repetitions such as $Y \to X \to Y$, Y's proof tree can be reused.

(b) If Y via (2) is proven with repetitions, neither X's nor Y's proof via (2) is retrieved at X and Y via (1) in the algorithm. Hence, X and Y are explored, guaranteeing a correct result.

The case of disproofs is similar. Thus the GHI solution guarantees that the draw-last case never happens.

■

For unproven nodes, the proposed GHI solution might compute incorrect proof and disproof numbers for df-pn, and incorrect heuristic values for $\alpha\beta$ search. However, the above theorems guarantee that (dis)proofs returned by this approach are always correct.

3.2.2 When Proofs and Disproofs Do Not Fit in Memory

In Section 3.2.1, in order to guarantee correctness of the GHI solution, proven and disproven nodes in the transposition table were assumed never to be replaced. One question is what happens if any entry is allowed to be replaced. Practical replacement and garbage collection schemes such as [11, 55] delete both proven and unproven nodes. The correctness of the proofs and disproofs saved in the transposition table is easily shown in the following way:

- If neither proofs nor disproofs are replaced, Theorems 3.2.1 and 3.2.2 guarantee that correct proofs and disproofs are saved in the transposition table.

- Even if replacement or garbage collection schemes delete proven and disproven table entries, every search result is determined by transposition table lookups and a search. The search result is therefore correct and is correctly saved in the transposition table.

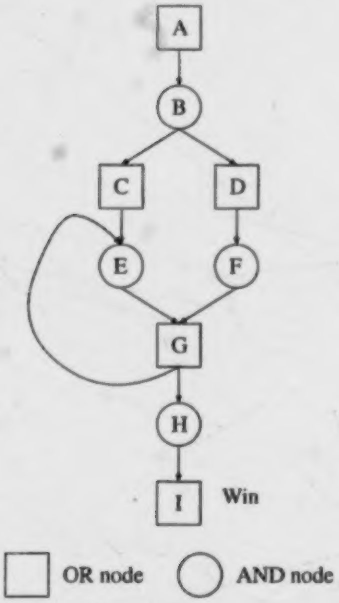


Figure 3.2: An example of constructing an incorrect proof tree with replacement schemes.

The above discussion ensures that using any replacement or garbage collection scheme still leads to correct values. However, incorrect proof or disproof trees are sometimes constructed, when proven or disproven nodes are discarded. Figure 3.2 shows an example. The first-player-loss-scenario is adapted in this figure and I is a win for the first-player. Assume that when either an OR node is proven or an AND node is disproven, one of the branches that constructs a proof or disproof tree is saved in the transposition table. Under these conditions, searching in the following order yields an incorrect proof tree:

- Explore $A \rightarrow B \rightarrow C \rightarrow E \rightarrow G \rightarrow H \rightarrow I$. Proofs are saved in the base entries of C, E, G, H, and I. The move leading to H is saved in G's entry, since G is proven by reaching H.
- Let G, H, and I be replaced by new entries.
- Explore $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G \rightarrow E$. Since a proof is saved in E's base table entry, G is also proven. G's table entry contains a move leading

to E, because G is proven by reaching E. Now, although G is a proven node, the reason why G is proven is incorrect. G must be proven by taking path $G \to H \to I$, not $G \to E \to G \to \cdots$.

Since this wrong proof construction is related to replacement schemes, the problem also occurs in previous work, such as Nagai's GHI solution [56]. Nagai suffered from a similar problem when his tsume-shogi solver performed re-searches to find other checkmating sequences [56]. This issue is also similar to Campbell's draw-last scenario, since the base table entry might implicitly include a repetition. However, the above discussion guarantees that there must be a valid proof or disproof in the base entry that is free from a repetition. The proof is never for the wrong reason, only the retrieval goes wrong.

To reconstruct a proof tree when using a replacement or garbage collection scheme that discards proven or disproven table entries, an algorithm that constructs a proof or disproof tree is required after the completion of the df-pn or $\alpha\beta$ search. For such a *proof tree reconstruction algorithm*, a small transposition table that is separate from the transposition table of the search is allocated. For the sake of simplicity, only the case of df-pn is explained. A similar approach can be applied to $\alpha\beta$. Let T be the transposition table for df-pn, and S be the small transposition table that contains a proof tree. Then, a simple re-search is performed to construct a proof tree in S. An analogous method can be used to construct a disproof tree. The reconstruction algorithm works as follows:

- Proven terminal nodes are saved in S.
- At an OR node, all branches that lead to proven AND children in T are traversed to construct a proof tree. If one of them succeeds, the proof tree is confirmed and saved in S.
- At an AND node, all children are traversed to construct a proof tree. If all of them succeed, the proof tree is saved in S.
- If no proven AND child is found in T at an OR node n, df-pn is called to prove n.

- If an incorrect repetition p at n, leading to a disproof of n via p, is detected, the reconstruction algorithm is terminated. Df-pn is invoked at the root node to construct another proof tree. Before df-pn is invoked, proof and disproof numbers of the nodes on the path of p are initialized to 1, and n via p is saved as a disproof in T. When df-pn returns a proof, the reconstruction algorithm is invoked again.
- The process is continued until a complete valid proof tree for the root is stored in S.

Since df-pn usually explores a much larger tree than a proof tree, the overhead of this algorithm is mostly small. However, the overhead incurred by the reconstruction algorithm is sometimes large. See Section 3.5 for experimental results.

3.3 Game-Specific Implementation Details

3.3.1 Go

All enhancements that will be explained in Chapter 5 are incorporated in the df-pn and $\alpha\beta$ implementations. The method that will be described in Chapter 4 is integrated with df-pn. $\alpha\beta$ performs iterative deepening, extends the search for forced moves, and searches the best move from a previous iteration first.

3.3.2 Checkers

8-piece endgame databases are incorporated in the df-pn and $\alpha\beta$ implementations. Scores obtained by database lookups are considered to be correct, because these scores are path independent. Simulation is not invoked for trees involving only database scores. The method that will be described in Chapter 4 is integrated with df-pn. The $\alpha\beta$ implementation performs a variable depth-first search with state-of-the-art enhancements.

3.4 Other Implementation Issues

One implementation problem has arisen in the GHI solution if the transposition table uses chaining to avoid collisions. When proving or disproving identical positions by repetitions via different paths, the GHI solution gives the identical transposition table key to these path-dependent positions. This property can cause a large number of chained twin table entries if the same position is proven or disproven via hundreds or thousands of different paths involving repetitions.

To solve the problem of long chains, two hash functions to compute transposition table keys are used. One key is an encoded position, the other key is an encoded path. The encoded position is called the *first hash key*, and the encoded path is called the *second hash key*. These hash keys are used as follows:

1. First hash keys are always used for unproven nodes.
2. Assume that A is a proven OR node or disproven AND node via path p, and m is a move that leads to a proof of A via p. The first hash key is computed and chained transposition table entries are traversed. If one of the chained table entries contains A via q and m is in the twin table entry of A via q, a (dis)proof of A via p is saved in the transposition table entry computed by the second hash key. Otherwise, A via p is saved in the table entry computed by the first hash key. This is reasonable because of the process of simulation: When simulation is invoked for A via another path, it needs to generate m by retrieving the table entry for A via p. m is generated by keeping track of all table entries which contain the first hash key of A. However, if A via q uses the first hash key to cache and contains m, A via p is not necessarily placed in the table entry with the first hash key. m can be generated by checking the table entry for A via q.
3. If A is a proven AND node or a disproven OR node via p, the move m in A via p is not used for simulation calls. Hence, A via p is not neces-

sarily stored by using the first hash key. The implementation therefore tries step 2 by assuming that m is constant. This technique randomly distributes (dis)proofs by repetitions over the transposition table.

3.5 Experiments

3.5.1 Setup

The df-pn and $\alpha\beta$ algorithms were applied to Go and checkers. The one-eye problem with situational super-ko in Go is a current-player-loss scenario. Checkers is a first-player loss scenario.

The experiments for programs ignoring and dealing with the GHI problem in Go were performed on an Athlon XP 2800 with a 300 MB transposition table. The experiments in checkers were performed on an Athlon 2400MP with a 300 MB transposition table. All proven and disproven nodes are saved in the transposition table in both programs. 162 positions in Go and 200 positions in checkers were prepared (see Appendix C). Each test suite was created in the following way:

- In contrast to full tsume-Go, for which many large collections of test problems are available, any specialized collection of one-eye problems could not be found in the literature. The current set of 81 test positions was created mainly by Martin Müller. The problems can be played for both colors going first, resulting in a total of 162 problems. All problems are of the following form: a group of the defender already has one safe eye, and is completely surrounded at a distance by safe stones of the attacker. The area in between forms the region, and the fate of the group of the defender depends on whether it can form a second eye in the region. Problems of this kind are also suitable for solution by a general tsume-Go solver, since making one eye is equivalent to solving the tsume-Go problem. The test set is available at `http://www.cs.ualberta.ca/~games/go/oneeye`. See Appendix C for a list of the positions. The problems include a mix of easy and hard problems.

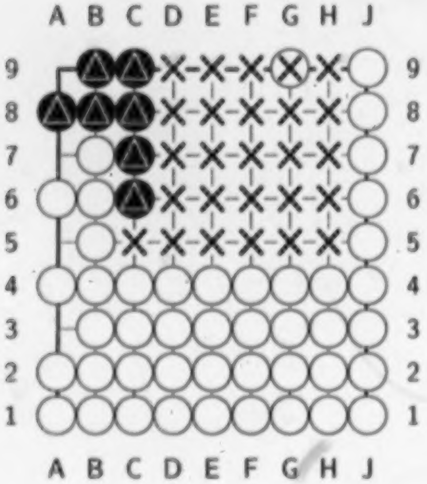


Figure 3.3: Example of a hard problem (Black to live).

Some problems are challenging only for one color playing first, and are very easy if the other color plays first. Some of the positions are hard to solve for current tsume-Go programs. For an example, see Figure 3.3.

- In checkers, test positions are randomly extracted from a partial proof tree of the White Doctor Opening. They were created by Neil Burch, Yngvi Björnsson, and Jonathan Schaeffer.

The time limit was set to 5 minutes per position in Go. In checkers the execution time was unstable because it is dominated by disk I/O to access the databases. Therefore, the execution time was not limited here. Instead, the checkers searches were limited to 20 million node expansions per position. Since df-pn can only return a binary answer, either proven or disproven, for a given position, it solves the question "win or no win." A draw is therefore considered a disproof in the test.

3.5.2 Results in Go

Tables 3.1 and 3.2 summarize the results for df-pn and $\alpha\beta$ in Go in terms of the number of problems solved and total node expansions. These statistics

Table 3.1: Performance comparison between ignoring and dealing with the GHI problem for df-pn in Go. All statistics are computed for 157 problems solved by both program versions.

Method used	Number of problems solved	Total nodes	Total time (sec)
IGNORE-GHI	$149 + 8$	82,002,480	1,757
HANDLE-GHI	157	84,084,752	1,975
Total problems	162	-	

Table 3.2: Performance comparison for $\alpha\beta$ in Go. All statistics are computed for 138 problems solved by both versions.

Method used	Number of problems solved	Total nodes	Total time (sec)
IGNORE-GHI	$136 + 2$	146,893,143	1,154
HANDLE-GHI	138	149,839,855	1,231
Total problems	162	-	

were collected from the two programs ignoring (IGNORE-GHI) and handling (HANDLE-GHI) the GHI problem. other approaches such as Nagai's could not be tested in Go, since they do not handle the current-player-loss scenario. Both IGNORE-GHI and HANDLE-GHI solve the same subset of problems. However, IGNORE-GHI gave incorrect proof or disproof trees for 8 positions in df-pn and for 2 positions in $\alpha\beta$. The incorrectness of (dis)proof trees was confirmed by re-checking (dis)proof trees computed by IGNORE-GHI. Although the scores returned by IGNORE-GHI were correct, it is important to have a scheme to handle the GHI problem, since GHI happens both in df-pn and $\alpha\beta$. Even if GHI does not appear in the final proof tree, it occasionally appears in the search. In the 157 problems solved, HANDLE-GHI in df-pn invoked simulation 70,003 times, explored 3,356,038 nodes by simulation, and discovered 4,230 flawed transposition table entries. In the 138 problems solved by $\alpha\beta$, HANDLE-GHI invoked simulation 8,500 times, explored 124,160 nodes by simulation, and detected 2,678 flawed entries. These numbers are conser-

Table 3.3: Performance comparison for df-pn in checkers. Node statistics are computed for the subset of 160 problems solved by all program versions.

Method used	Problems solved	Total nodes
IGNORE-GHI	$138 + 26$	188,422,395
HANDLE-GHI	166	187,297,631
NAGAI	163	212,237,026
Total problems	200	-

vative, because some incorrect proofs or disproofs may have been stored but never retrieved. As in Campbell and Breuker's papers [13, 12], these numbers confirm that GHI occurs more frequency in df-pn than in $\alpha\beta$. However, the results show that it is still necessary to cure GHI in $\alpha\beta$.

Figures 3.4 to 3.7 compare the performance between IGNORE-GHI and HANDLE-GHI for each problem. In Figures 3.4 and 3.6, the node expansions of HANDLE-GHI are plotted on the X-axis against IGNORE-GHI on the Y-axis on logarithmic scales. A point above the diagonal means that HANDLE-GHI performed better. Similarly, in Figures 3.5 and 3.7, the execution time spent by HANDLE-GHI is plotted on the X-axis against IGNORE-GHI on the Y-axis on logarithmic scales. The results show that HANDLE-GHI can avoid the GHI problem with negligible overhead in terms of node expansions. In terms of execution time, IGNORE-GHI solves problems slightly faster than HANDLE-GHI in df-pn and almost as quickly as HANDLE-GHI in $\alpha\beta$. In total, HANDLE-GHI explored 2.5% extra nodes and needed 12.4% extra time in df-pn. In $\alpha\beta$, HANDLE-GHI explored 2.0% extra nodes, and needed 6.7% extra time. This is a small price to pay for guaranteeing correctness.

3.5.3 Results in Checkers

Table 3.3 gives the results for df-pn in checkers. Nagai's solution (NAGAI) to the GHI problem [56] is additionally implemented. Of the 200 problems in the test set, 160 problems are solved by all methods. Table 3.4 summarizes the extra problems solved by each method. HANDLE-GHI solved 3 extra problems that IGNORE-GHI did not solve. IGNORE-GHI solved 164 problems, includ-

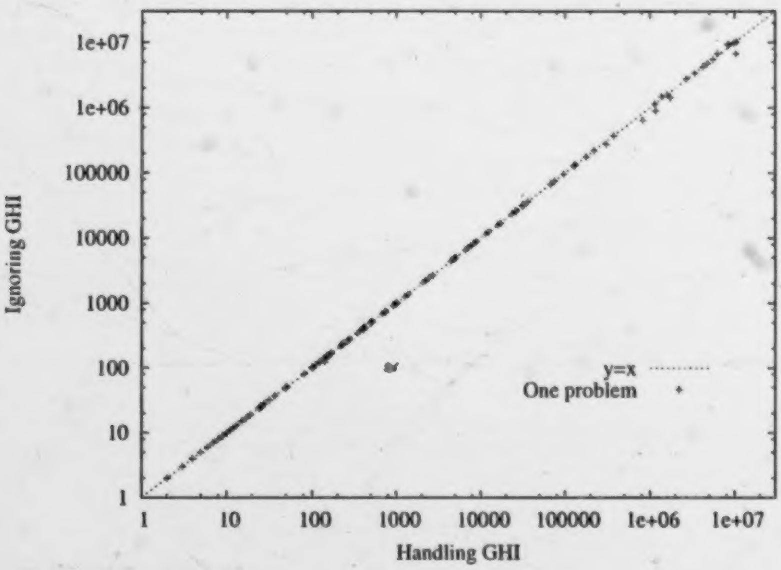


Figure 3.4: Node expansions for problems solved by both HANDLE-GHI and IGNORE-GHI in df-pn in Go.

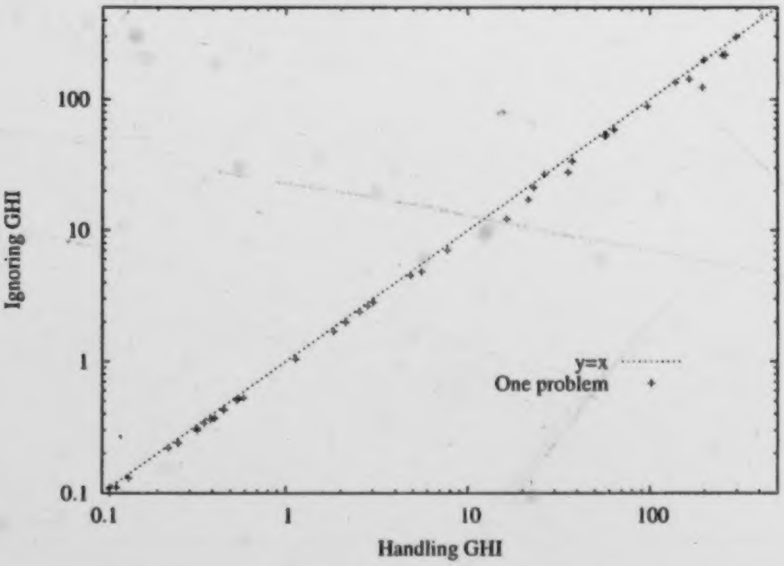


Figure 3.5: Execution time for problems solved by both HANDLE-GHI and IGNORE-GHI in df-pn in Go.

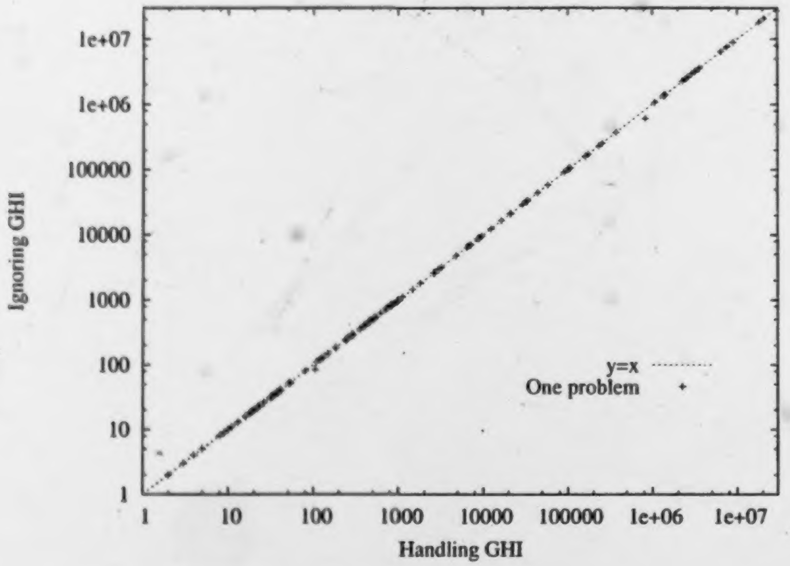


Figure 3.6: Node expansions for problems solved by both HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in Go.

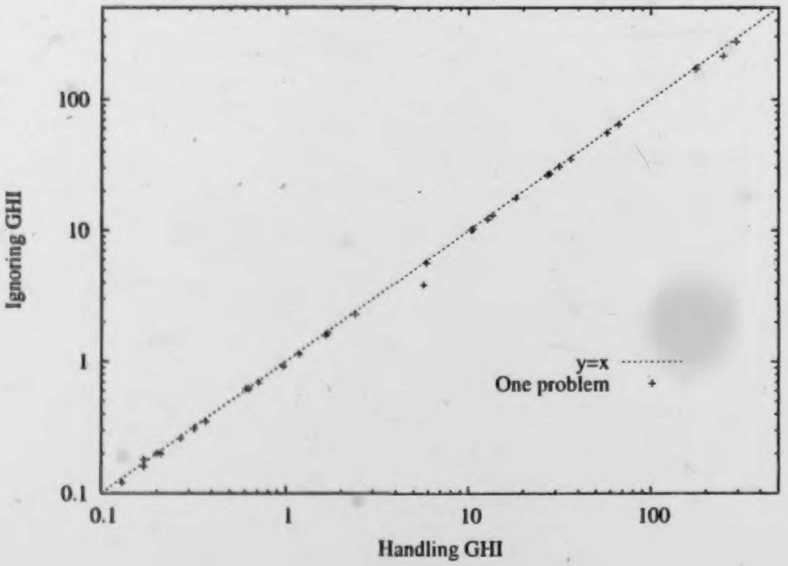


Figure 3.7: Execution time for problems solved by both HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in Go.

Table 3.4: Extra problems solved by each method. See Appendix C.2 for a listing of the 200 problems.

Method	Problem Id
IGNORE-GHI	17 62 76 136
HANDLE-GHI	55 62 76 78 87 136
NAGAI	55 87 156

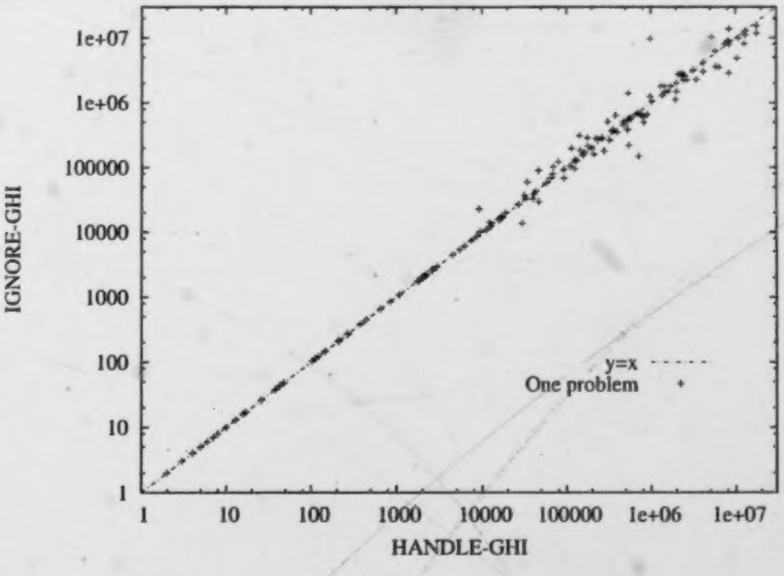


Figure 3.8: Comparison of node expansions between HANDLE-GHI and IGNORE-GHI in df-pn in checkers.

ing one problem which was not solved by HANDLE-GHI. However, IGNORE-GHI generated incorrect disproofs in 26 cases, including the one extra problem solved only by IGNORE-GHI. Figure 3.8 compares node expansions for the problems solved by both IGNORE-GHI and HANDLE-GHI. Figure 3.8 indicates that the GHI solution does not degrade the performance. HANDLE-GHI even explored 0.6% less total nodes than IGNORE-GHI.

Compared with Nagai's method, HANDLE-GHI solved 4 extra problems that NAGAI did not solve. On the other hand, NAGAI solved one more problem unsolved by HANDLE-GHI. Although both approaches solved problems correctly, the GHI solution can be applied to various algorithms, whereas

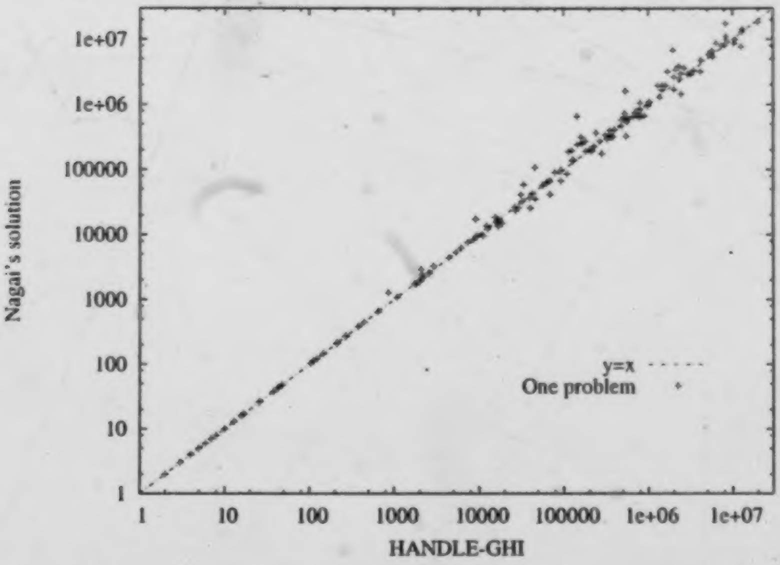


Figure 3.9: Node expansions for problems solved by HANDLE-GHI and NAGAI in df-pn in checkers.

Table 3.5: Performance comparison for $\alpha\beta$ in checkers. Node statistics are computed for the subset of 119 problems solved by both program versions.

Method used	Problems solved	Total nodes
IGNORE-GHI	109 + 12	244,508,069
HANDLE-GHI	120	229,763,174
Total problems	200	-

Nagai's method can be incorporated only to df-pn. Figure 3.9 plots node expansions for each problem. The plot is slightly above the diagonal line. In terms of total nodes for problems solved by all versions, NAGAI explored 13.3% more nodes than HANDLE-GHI.

Simulation again detects flawed transposition table entries. In the 166 solved problems, HANDLE-GHI invoked simulation 720,684 times with 5,787,658 node expansions and discovered 332,362 flawed transposition table entries. These numbers confirm that the GHI problem occurs in the search. These incorrect results sometimes appear in the final disproof trees of IGNORE-GHI.

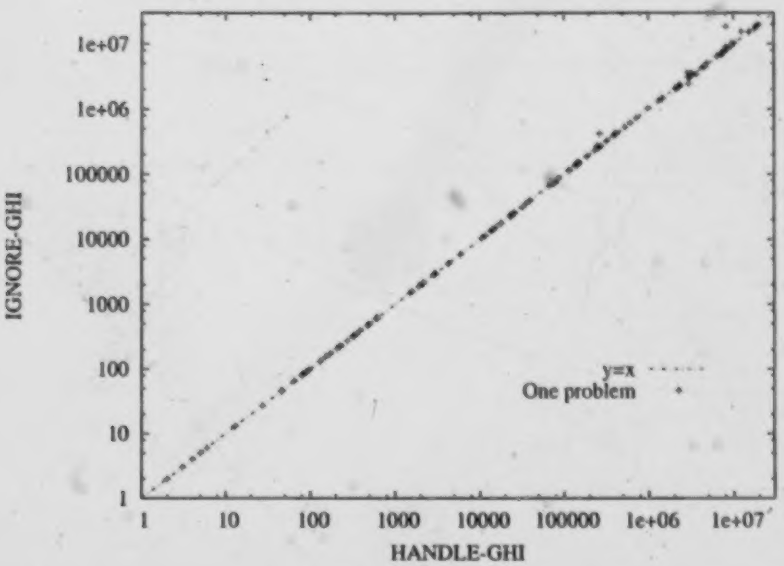


Figure 3.10: Node expansions for problems solved by HANDLE-GHI and IGNORE-GHI in $\alpha\beta$ in checkers.

Table 3.5 shows the results for $\alpha\beta$. IGNORE-GHI solved 2 extra positions that were not solved by HANDLE-GHI, whereas HANDLE-GHI solved one problem unsolved by IGNORE-GHI. IGNORE-GHI returned incorrect disproofs for 12 positions, including the two positions 'solved' only by IGNORE-GHI. Figure 3.10 plots node expansions for problems solved by both versions. Most results are close to the diagonal line, which implies that HANDLE-GHI incurs a negligible overhead. In terms of total nodes, HANDLE-GHI even explored 6.4% less nodes than IGNORE-GHI.

In the 121 problems solved, HANDLE-GHI invoked simulation 48,699 times with 81,772 node expansions and detected 29,852 flawed entries. These numbers indicate that the GHI problem occurs in $\alpha\beta$ in checkers, but much less frequently than in df-pn.

In conclusion, since the GHI problem happens both in df-pn and $\alpha\beta$ in checkers, it is dangerous to ignore. Because the GHI solution not only incurs low overhead but also always returns correct answers, it is a worthwhile addition to any search engine susceptible to GHI.

The proposed solution could be compared with Breuker's BTA. However, BTA needs an explicit graph representation, and complicated operations to deal with repetitions. This causes a problem when BTA uses up available memory. On the other hand, the approach proposed in this Chapter does not need any explicit graph representation. Transposition table entries can be replaced when the table becomes full. Breuker's scheme to detect real draws is specific to the first-player-loss scenario, and could be added to the framework.

3.5.4 Results with Limited Memory

In the previous experiments, proofs and disproofs are never deleted from the transposition table. In this section, the performance of the algorithms was measured when using a garbage collection scheme. Nagai's SmallTreeGC [55], which discards transposition table entries containing small subtrees was implemented. Two versions were implemented in Go: one that always keeps proven and disproven table entries (*KEEP-PROOF*), and one that can discard proven and disproven entries (*DISCARD-PROOF*). The size of transposition table was varied from 5 MB to 30 MB for df-pn and $\alpha\beta$. The size of a table entry is 48 byte in both df-pn and $\alpha\beta$. For example, a 30 MB transposition table contains 655,360 entries in df-pn and $\alpha\beta$. 10 hard problems which do not fit in memory with a 30 MB transposition table were chosen. Tables 3.6 and 3.7 list the problems used in the experiments. Node expansions and execution time in the tables are from data in KEEP-PROOF with a 300 MB transposition table. Separate test sets are used for df-pn and $\alpha\beta$, because problems solved by $\alpha\beta$ are usually easy for df-pn. The time limit was set to 400 seconds per position.

Figures 3.11 compares the solving ability of the two methods for df-pn. The size of the transposition table is plotted on the X-axis against the number of problems solved by each method on the Y-axis. DISCARD-PROOFS works better than KEEP-PROOFS. However, the test set used in the experiments contains only very hard problems, to be solved with a small amount of memory. Most problems that are excluded from these experiments are easy for both KEEP-PROOFS and DISCARD-PROOFS, because most of them fit in

Table 3.6: List of problems used for df-pn.

Problem name	Color to play	Node expansions	Execution time (sec)
oneeyee.1.sgf	Black	1,732,845	35.65
oneeyee.2.sgf	White	5,557,002	164.77
oneeyee.3.sgf	Black	1,601,033	37.52
oneeyee.3.sgf	White	8,293,193	198.46
oneeyee.4.sgf	White	10,288,160	194.26
oneeyee.5.sgf	Black	6,277,938	138.65
oneeyee.7.sgf	Black	4,771,253	97.10
oneeyee.8.sgf	White	10,550,455	258.69
oneeyee.9.sgf	White	2,711,384	57.70
oneeyee.11.sgf	White	9,783,406	250.71

Table 3.7: List of problems used for $\alpha\beta$.

Problem name	Color to play	Node expansions	Execution time (sec)
oneeyec.1.sgf	White	2,604,146	18.16
oneeyec.6.sgf	White	29,897,113	248.99
oneeyec.7.sgf	Black	6,449,003	36.36
oneeyec.9.sgf	Black	18,090,794	174.99
oneeyed.2.sgf	White	8,926,242	66.58
oneeyed.6.sgf	Black	20,698,195	177.46
oneeyee.1.sgf	White	7,540,735	57.42
oneeyee.6.sgf	White	3,162,111	27.12
oneeyee.7.sgf	White	34,758,867	293.45
oneeyee.10.sgf	White	3,571,438	31.36

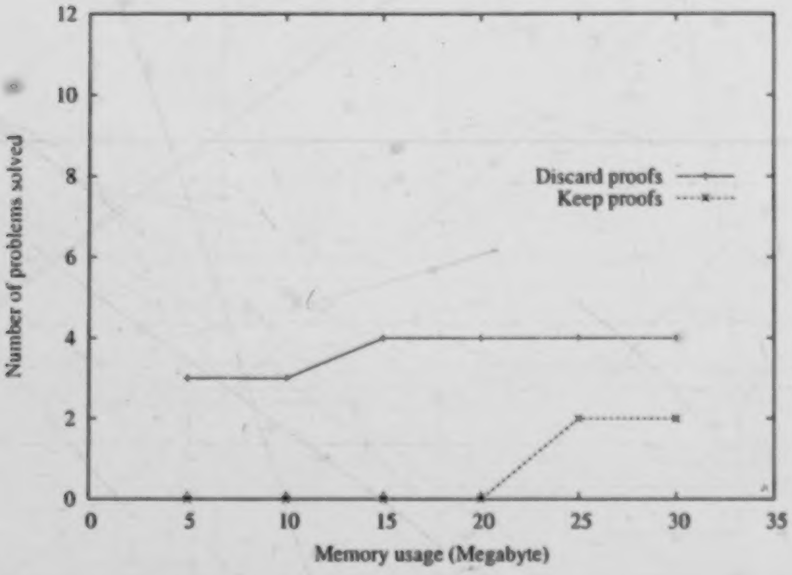


Figure 3.11: Comparison of the solving ability of DISCARD-PROOFS and KEEP-PROOFS in df-pn.

memory. All problems solved by KEEP-PROOFS are solved by DISCARD-PROOFS. Moreover, DISCARD-PROOFS solves more problems even with a 5 MB transposition table than KEEP-PROOFS with a 30 MB transposition table. In KEEP-PROOFS, proofs and disproofs are saved in most of the table entries, causing the performance to degrade. For example, with a 30 MB transposition table entries, 99.8% (more than 654,300 entries out of 655,360) contained proven or disproven nodes for the 8 unsolved problems.

Since DISCARD-PROOFS may construct a wrong proof tree, the proof tree reconstruction algorithm in Section 3.2.2 is invoked after df-pn returns a yes/no answer. Tables 3.8 and 3.9 show the overhead to reconstruct proof trees for solved problems. Total node expansions are computed as the sum of the numbers of nodes explored by df-pn and the proof tree reconstruction algorithm. The overhead is usually a few percent. However, in one problem, oneeyed.9.sgf, the solver spent more than 20% of node expansions for constructing a proof tree. Df-pn created an incorrect proof tree in the final proof tree as in Figure 3.2. The proof reconstruction algorithm detected it and

Table 3.8: List of problems solved by df-pn with a 5 MB transposition table.

Problem name	Color to play	Total Node expansions	Node Expansions for Proof tree reconstruction
oneeyee.1.sgf	Black	7,812,083	475,536 (6.1%)
oneeyee.3.sgf	Black	6,289,650	446,501 (7.1%)
oneeyee.9.sgf	White	8,662,833	1,869,667 (21.6%)

Table 3.9: List of problems solved by df-pn with a 30 MB transposition table.

Problem name	Color to play	Node expansions	Node Expansions for Proof tree reconstruction
oneeyee.1.sgf	Black	1,846,981	122,775 (6.6%)
oneeyee.3.sgf	Black	1,587,828	104,309 (6.6%)
oneeyee.7.sgf	Black	8,381,461	217,358 (2.6%)
oneeyee.9.sgf	White	3,998,017	969,255 (24.2%)

needed to construct a correct proof tree. However, considering the fact that oneeyee.9 is not solved by KEEP-PROOFS, this is still a price worth paying to achieve a better solving ability. Also, since the previous garbage collection or replacement schemes such as [11] and [55] already deleted proven and disproven nodes, these algorithms also need some methods to reconstruct a proof tree. Moreover, these previous garbage collection or replacement schemes deleted proofs and disproofs without considering the GHI problem.

Figure 3.12 compares the number of problems solved by DISCARD-PROOFS and KEEP-PROOFS for $\alpha\beta$. The problems used for $\alpha\beta$ are easy for df-pn. All problems solved by KEEP-PROOFS are solved by DISCARD-PROOFS. The benefit of discarding proven and disproven transposition table entries are more vividly shown in this graph. Again, if the size of the transposition table is small, more than 99% of the transposition table entries of KEEP-PROOFS are filled out by proofs and disproofs. KEEP-PROOFS thus suffers from the degradation in performance.

Tables 3.10 and 3.11 show the overhead of reconstructing proof trees in $\alpha\beta$. Although the overhead of re-searches to reconstruct proof trees is usually

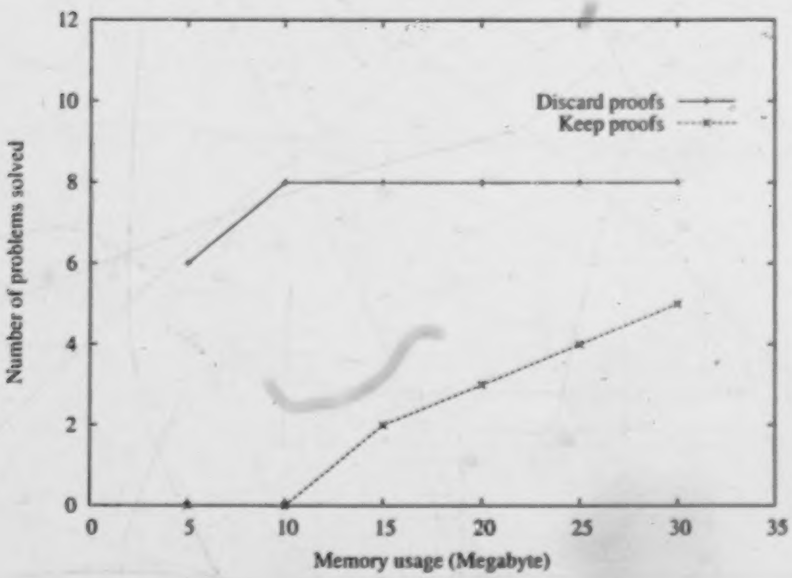


Figure 3.12: Comparison of the solving ability of DISCARD-PROOFS and KEEP-PROOFS in $\alpha\beta$.

small, a large amount of search is occasionally performed. In particular, more than half of all node expansions are needed for reconstructing a proof tree in oneeyee.1.sgf in Table 3.10. This position was not solved by KEEP-PROOF. Again, this is the price to pay for deleting proofs and disproofs from the transposition table.

3.5.5 An Example of GHI in Go with the SSK Rule

A position that suffered from the GHI problem with the SSK rule was found when the solver that ignores GHI was implemented (see Figure 3.13 and Figure 3.14(a)). First the solver explored the position by tracing the move sequence in Figure 3.14(b). Path **C7** → (1) → (3) → **A8** → (4) in Figure 3.13 corresponds to this move sequence. Then, Black cannot play at **A8**, because this move leads back to the position after move 4, shown in Figure 3.14(c). Based on this result, a win for White is saved in the table entry for position (c). However, after the sequence in Figure 3.14(d), position (c) is no longer a win for White. When the solver traced move sequence (d), it first reached (e) and then encountered

Table 3.10: List of problems solved by $\alpha\beta$ with a 5 MB transposition table.

Problem name	Color to play	Node expansions	Node Expansions for Proof tree reconstructions
oneeyec.1.sgf	White	3,429,461	577,095 (16.8%)
oneeyec.7.sgf	Black	10,802,300	74,792 (0.7%)
oneeyed.2.sgf	White	15,858,823	1,307,200 (8.2%)
oneeyee.1.sgf	White	29,222,605	15,810,038 (54.1%)
oneeyee.6.sgf	White	4,102,290	462,400 (11.2%)
oneeyee.10.sgf	White	7,751,578	2,194,344 (28.3%)

Table 3.11: List of problems solved by $\alpha\beta$ with a 30 MB transposition table.

Problem name	Color to play	Node expansions	Node Expansions for Proof tree reconstructions
oneeyec.1.sgf	White	2,906,373	296,307 (10.2%)
oneeyec.7.sgf	Black	7,185,485	38,439 (0.5%)
oneeyec.9.sgf	Black	29,992,774	783,018 (2.6%)
oneeyed.2.sgf	White	10,151,792	549,229 (5.4%)
oneeyed.6.sgf	Black	31,567,671	1,235,226 (3.9%)
oneeyee.1.sgf	White	11,937,308	2,179,076 (18.3%)
oneeyee.6.sgf	White	3,286,143	652,258 (17.0%)
oneeyee.10.sgf	White	4,190,165	594,050 (14.2%)

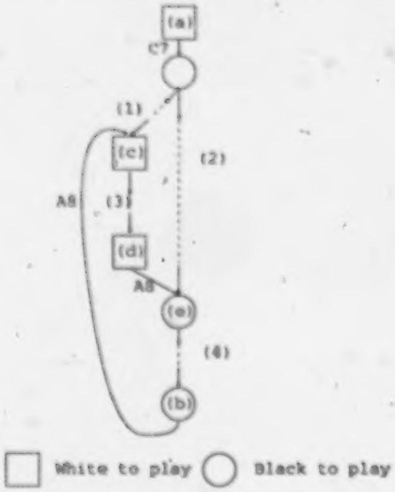


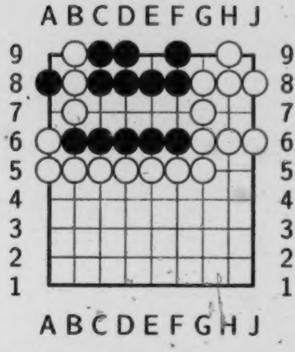
Figure 3.13: Repetitions in the move sequences in Figure 3.14.

(c). However, after the sequence (d), White cannot play a move at **A8**, since it leads back to position (e). Path **C7** → (2) → (4) → **A8** → (3) in Figure 3.13 stands for this sequence of moves. The correct result for position (c) via Figure 3.14(d) is a win for Black.

Remark that White **15 at 7** in Figure 3.14(b) would be better than **15 at 1**, and White **15 at 5** would be better in Figure 3.14(d), because then White can win without repetition in both cases. However, even if adding more game-specific knowledge to the one-eye solver could reduce the number of such cases, there is no general way to always find a non-repetition proof first.

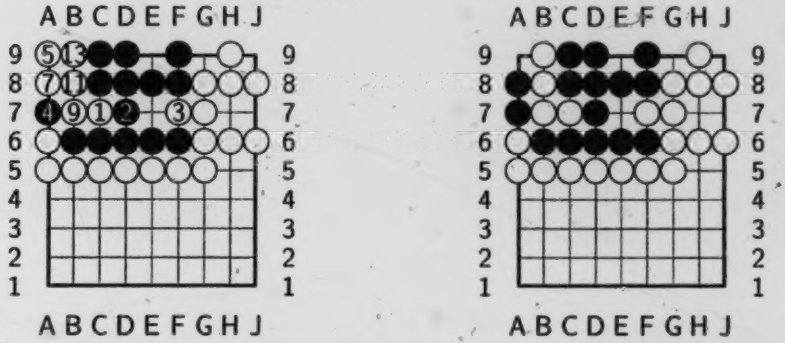
3.6 Conclusions

This chapter presented a framework to solve an important open problem raised by [60] 20 years ago. This approach incurs very small overhead and is applicable to algorithms such as df-pn and $\alpha\beta$. The solution to the GHI problem is concluded to be both practical and general. Additionally, the GHI solution guarantees a correct yes/no answer even if the proof does not fit in memory. To reconstruct a correct proof or disproof tree, a re-search must be performed. Although the proof reconstruction algorithm usually incurs a small overhead,



The original position: White to play.

(a)

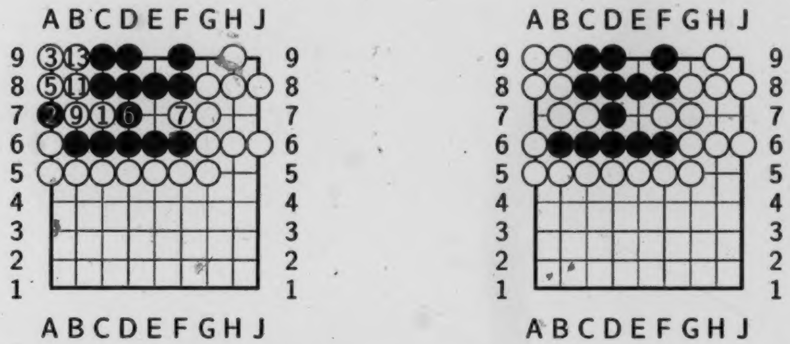


6 at 4, 8 at 4, 10 at 7, 12 at 5, 14 at 4, 15 at 1.

(b)

White to play after move 4.

(c)



4 at 2, 8 at 2, 10 at 5, 12 at 3, 14 at 2,
15 at 1, 16 at 5, 17 at 3, 18 at 2.

(d)

Black to play after move 7.

(e)

Figure 3.14: An instance of the GHI problem in Go.

a large amount of re-searches is sometimes performed. However, since discarding proofs and disproofs allows the solvers to run in limited memory, it is worth discarding them.

Chapter 4

Depth-First Proof-Number Search with Repetitions

When standard df-pn was applied to the one-eye problem in Go, it could not solve some of the easy problems. The algorithm has a fundamental problem when applied to a domain with repetitions. The existence of the same problem was also confirmed in checkers. This chapter addresses the problems of df-pn related to repetitions and presents a solution, called df-pn(r).

4.1 Problem Description

Figure 4.1 shows an example of the problem of the standard df-pn algorithm. Assume F is unknown, then the df-pn algorithm computes $\mathbf{pn}(E) = \mathbf{pn}(A) + \mathbf{pn}(F)$. Hence, $\mathbf{pn}(E)$ is larger than $\mathbf{pn}(A)$. Df-pn's termination condition is (see Figure 2.4 in Chapter 2):

$$n.\phi \leq \Delta\mathrm{Min}(n) \mid\mid n.\delta \leq \Phi\mathrm{Sum}(n)$$

Usually the threshold of the proof number is only a little bit larger than $\mathbf{pn}(A)$ when exploring A's subtree in df-pn. Therefore, assuming that df-pn reaches E, df-pn exceeds the proof number threshold, stops expanding and updates A's proof number to $\mathbf{pn}(E) = \mathbf{pn}(A) + \mathbf{pn}(F)$. Even if E is chosen again in a later iteration, this phenomenon continues. The search exceeds the threshold at E and F is never explored. These repetitions often happen in Go, because passes are allowed. Two consecutive passes lead back to the same

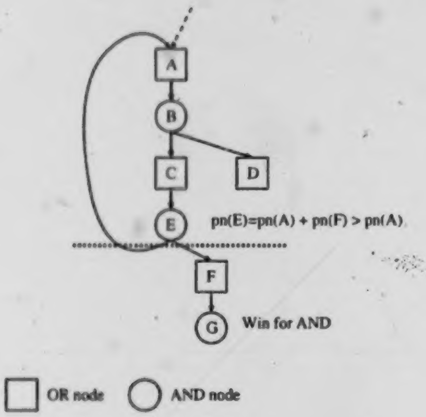


Figure 4.1: A problem with repetitions in df-pn.

position in a short loop.

One might argue that this problem may be solved by The solution to the GHI problem explained in the last chapter. If a repetition is detected in the GHI solution, a proof or disproof via a path is immediately saved. Therefore, the above simple example is not a problem for df-pn with the GHI solution. However, the problem does occur even with the GHI solution. Figure 4.2 shows an example. Let us call df-pn with the GHI solution *df-pn(GHI)*. A detailed proof that df-pn(GHI) cannot solve this graph is given in Appendix A. In this figure, when reaching O via $A \rightarrow C \rightarrow G \rightarrow J \rightarrow O$, the threshold of the disproof number is $\mathbf{dn}(O)$. Df-pn(GHI) stops expanding and updates proof and disproof numbers above O. Moreover, when df-pn(GHI) reaches N via $A \rightarrow C \rightarrow F \rightarrow I \rightarrow L \rightarrow N$ and $\mathbf{dn}(N)$ is computed, $\mathbf{dn}(O)$, which is a bit larger than $\mathbf{dn}(N)$, is added to $\mathbf{dn}(N)$. The threshold of the disproof number is $\mathbf{dn}(N)$ and no expansion occurs below N. Hence, although Q must be reached to prove that A is a win, df-pn(GHI) never explores Q.

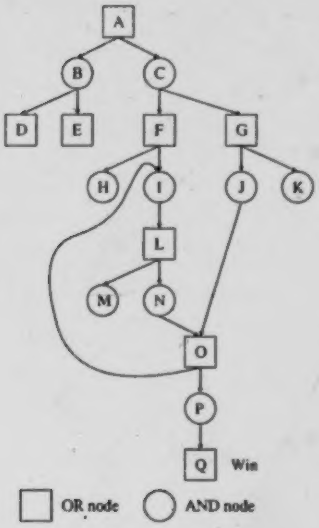


Figure 4.2: An example in which df-pn loops forever even with the GHI solution.

4.2 Computing Proof and Disproof Numbers in Domains with Repetitions

The new improved version of df-pn is called *df-pn(r)*. Df-pn(r) incorporates the GHI solution in the last chapter, and modifies the scheme for computing proof and disproof numbers in the presence of repetitions. Adding proof numbers from an ancestor to a node seems intuitively bad, since it leads to double-counting of the leaf nodes below. In the proposed solution to this problem, the children of a node are classified into two types. A field *minimal distance (md)* of a node n is initially set to the length of the shortest path from the root to n, the depth of n in the search tree. The notion of minimal distance is later extended based on search results. A child n_i is called *normal* if $n_i.md > n.md$, and *old* if $n_i.md \leq n.md$. Among the children $n_1 \cdots, n_k$ of n, let $n_1 \cdots, n_l$ $(1 \leq l \leq k)$ be the normal and $n_{l+1}, \cdots, n_k$ the old children. The computation of proof and disproof numbers is modified in the following way:

$$n.\phi = \min_{1 \leq i \leq k} n_i.\delta$$

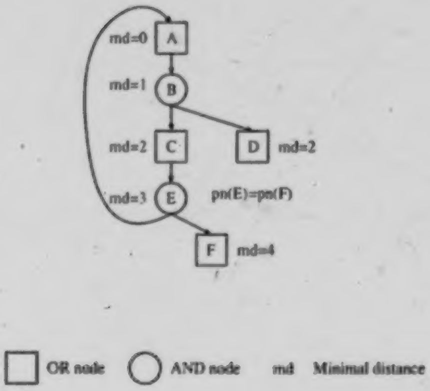


Figure 4.3: Df-pn with minimal distance *md*.

$$n.\delta = \begin{cases} \sum_{i=1}^{l} n_i.\phi & (\text{if } \sum_{i=1}^{l} n_i.\phi \neq 0) \\ \max_{l+1 \le i \le k} n_i.\phi & (\text{if } \sum_{i=1}^{l} n_i.\phi = 0) \end{cases}$$

In this computation scheme, the maximum (dis)proof is taken if only old children remain unproven. The max operation is reasonable since it preserves the following basic properties of proof and disproof numbers:

- If at least one child is (dis)proven at an OR (AND) node n, n is (dis)proven.
- If all children are (dis)proven at an AND (OR) node n, n is (dis)proven.

Figure 4.3 illustrates an example of computing proof numbers. If F is neither proven nor disproven, then F's proof number cannot be 0. Therefore A is ignored to compute E's proof number, since A is an old child.

When a node has only old children, since all normal (and possibly some old) children have been solved, that node itself must be considered old, since now there is no way to prove or disprove it without exploring old nodes. Therefore, the *md* field of that node must be updated. It is set to the minimum of the *md* fields of the currently unsolved old children.

Figure 4.4 presents pseudo-code that computes (dis)proof numbers and updates *md*. Each transposition table entry contains an extra field for *md*.

```
void MID(node &$n$) {
  ...
  // Terminal node
  if (IsTerminal($n$)) {
    if (WinForCurrentNode($n$)) {
      ...
    } else{
      ...
    }
    TTstore($n$,$n.\phi$,$n.\delta$, $n.md$);
    return;
  }
  ...
  // Iterative deepening
  $\phi_n$ = $\Phi$Sum($n$);
  while ($n.\phi > \Delta$Min($n$)  &&
          $n.\delta > \phi_n$)  {
    ...
    MID($n_c$); $\phi_n$ = $\Phi$Sum($n$);
  }
  // Store search results
  $n.\phi$ = $\Delta$Min($n$); $n.\delta = \phi_n$;
  TTstore($n$,$n.\phi$,$n.\delta$, $n.md$);
}
```

```
// Compute sum of $\phi$ of $n$'s children
int $\Phi$Sum(node $n$) {
  int $sum$ = 0, $max$ = 0;
  int $md$ = $n.md$;
  for (each child $n_{child}$ of $n$) {
    TTlookup($n_{child}$,$\phi$,$\delta$,$md_{child}$);
    if ($n.md < md_{child}$)
      // Normal child
      $sum = sum + \phi$;
    else{
      // Old child
      $max$ = max($max$, $\phi$);
      $md$ = min($md$,$md_{child}$);
    }
  }
  if ($sum$ != 0)
    return $sum$;
  else {
    // Update $md$ if only old children
    // exist
    $n.md = md$;
    return $max$;
  }
}
```

Figure 4.4: Pseudo-code of the df-pn(r) algorithm.

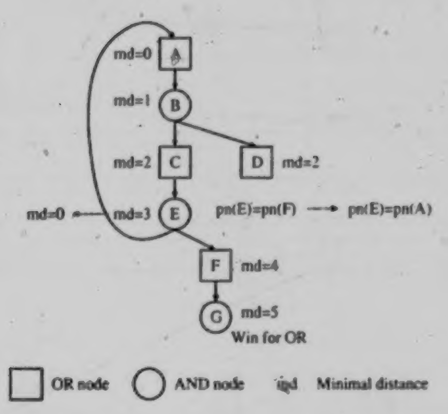


Figure 4.5: Updating E's minimal distance.

TTlookup retrieves information on *md*, and *TTstore* saves *md* in the transposition table. If no unproven normal children of exist, n's minimal distance is set to the smallest *md* of the children.

Figures 4.5 and 4.6 depict an example of updating *md*. In this figure, assuming that G is proven, E now has only an old child to explore, because F is also proven. In that case E's minimal distance is updated to A's distance, and $\mathbf{pn}(E)$ becomes $\mathbf{pn}(A)$. Further, $C.md$ is set to $E.md$, since E is now an old child of C and the only child to explore (see Figure 4.6). As a result, $\mathbf{pn}(C)$ is now ignored in the computation of $\mathbf{pn}(B)$, since C has become an old child.

Dealing with over-counting proof numbers caused by repetitions was essential to make df-pn work, in domains such as Go and checkers. Note that Nagai achieves impressive results with his tsume-shogi solver, and described the GHI problem, which returns incorrect results involving cycles [56]. However, the problem addressed above was not described in his papers. One possibility is that although the same problem could happen in shogi, it might happen much less often. On the other hand, search in Go can easily return to identical states, for example by consecutive pass moves. Another possibility is that this problem tends to happen less frequently with additional search enhancements. Because Nagai's tsume-shogi solver is enhanced with a lot of domain-dependent

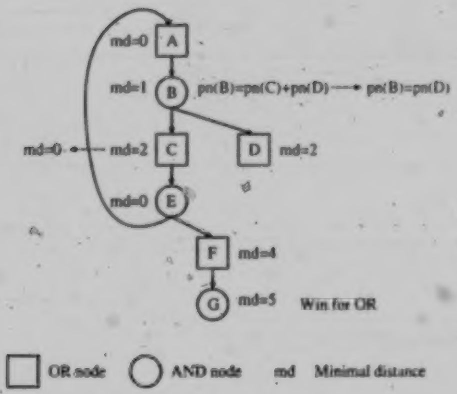


Figure 4.6: Computing C's minimal distance.

knowledge, it might not occur in his case in practice. However, the existence of this problem in shogi was confirmed by personal communication with Tsuruoka and Maruyama of team Gekisashi. The existence of the problem in shogi was confirmed when implementing a tsume-shogi solver based on df-pn for the IS shogi project [34]. Sakuta found that df-pn did not work better than PDS [55] in his tsume-shogi solver, and gave as possible explanation the occurrence of cycles [66].

4.3 Experimental Results

4.3.1 Setup

The df-pn algorithm was applied to Go and checkers. The experiments for programs ignoring and dealing with computing proof and disproof numbers with repetitions are performed. *Old df-pn* is the plain df-pn solver which does not incorporate the proposed method, whereas df-pn(r) uses it. Both versions already include the GHI solution explained in the last chapter. This implies that trivial repetitions are already naturally handled by the GHI solution. Moreover, one refinement added to all solvers is that if one of the large preset thresholds $\infty - 1$ (see Section 3.1.6) is exceeded at a child n_c of n, the solvers give up solving n_c and try n's other children. The same experimental

Table 4.1: Performance comparison between old df-pn and df-pn(r) in Go.

	Number of problems solved	Total time (sec) 150 problems	Total nodes expanded 150 problems	Nodes expanded per second
Old df-pn	150	281	11,297,348	40,256
Df-pn(r)	157	251	9,783,406	39,024
Total problems	162	-	-	-

Table 4.2: Performance comparison between old df-pn and df-pn(r) in checkers.

	Number of problems solved	Total nodes expanded 115 problems
Old df-pn	115	91,206,147
Df-pn(r)	166	25,793,405
Total problems	200	-

conditions explained in Section 3.5.1 were used in the experiments.

4.3.2 Results

Tables 4.1 and 4.2 compare the solving abilities of the version with and without the method for computing proof and disproof numbers related to repetitions. More problems are solved by df-pn(r). Moreover, all problems solved by old df-pn are also solved by df-pn(r) in both games. This indicates the superiority of df-pn(r).

Figures 4.7 and 4.8 present statistics for the problems solved by both versions in Go. Logarithmic scales were used to plot each problem. The number of nodes explored by df-pn(r) is plotted on the X-axis against old df-pn on the Y-axis. A point above the diagonal means that df-pn(r) performed better. Since most of the plots stay around the diagonal line in terms of execution time and node expansions, adding df-pn(r) does not degrade the performance of the solver.

Figures 4.9 and 4.10 present statistics on the problems solved only by

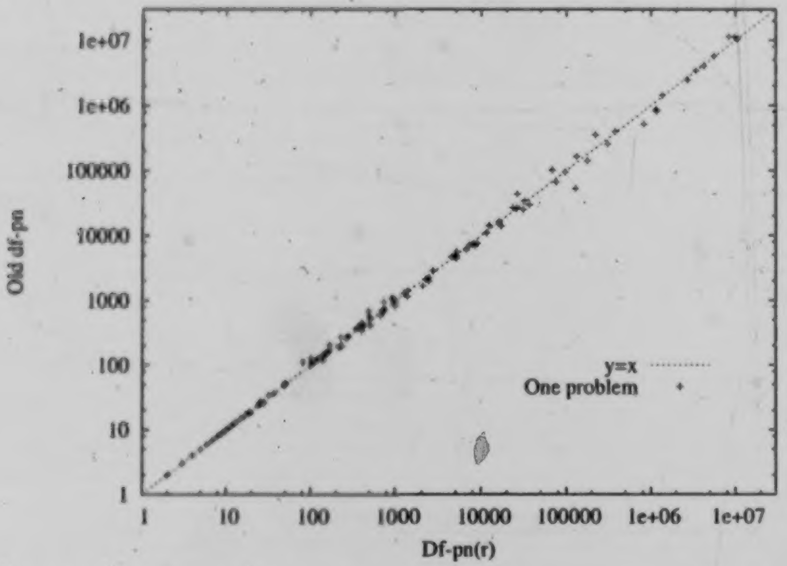


Figure 4.7: Node expansions for problems solved by both versions in Go.

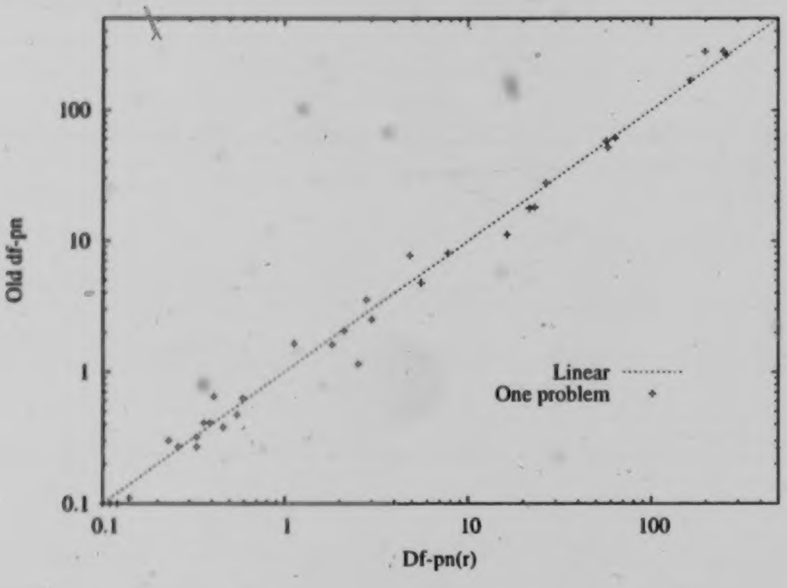


Figure 4.8: Execution time for problems solved by both versions in Go.

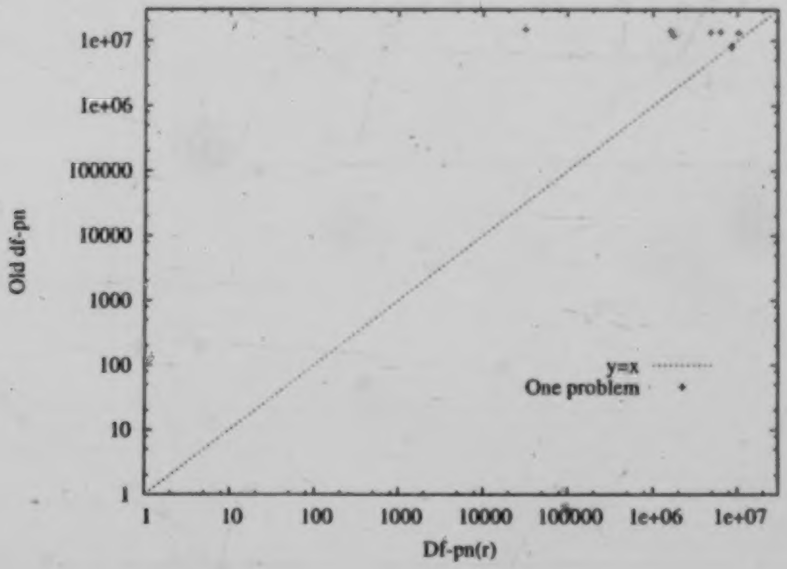


Figure 4.9: Node expansions for problems solved only by df-pn(r) in Go.

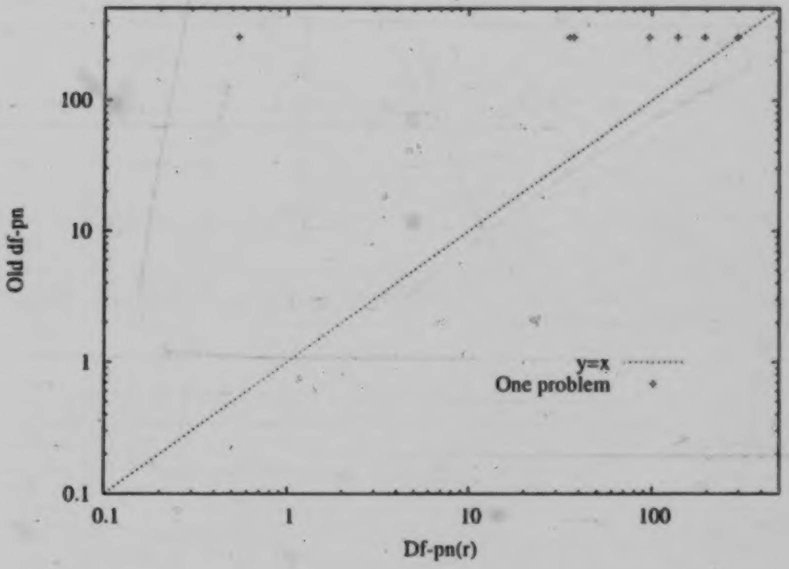


Figure 4.10: Execution time for problems solved only by df-pn(r) in Go.

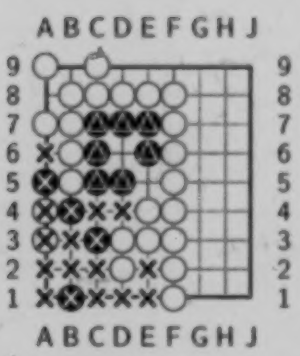


Figure 4.11: An example easily solved by df-pn(r) in Go (oneeyed.8.sgf, Black to live at **A2**).

df-pn(r). Again, logarithmic scales are used to plot each problem. In Figure 4.9, the number of nodes explored for 5 minutes by old df-pn is plotted on the Y-axis. In Figure 4.10, 5 minutes, the time used by old df-pn, is plotted on the Y-axis. The difficulties of the problems unsolved by old df-pn vary from easy to hard. These figures confirm that some problems that would have been easily solved become unsolvable with old df-pn. For example, df-pn(r) solved the position in Figure 4.11 with only 31,710 nodes in 0.54 seconds, whereas old df-pn explored 14,800,130 nodes but was still unable to solve it. In this position, old df-pn encounters the problem of computing proof numbers at an early stage (after about 30,000 node expansions) and loops forever.

In checkers, node expansion statistics for the problems solved by both versions are presented in Figure 4.12. The results are similar to Go. Both methods explore a similar amount of nodes. However, old df-pn occasionally expands much more nodes, as shown in Figure 4.12. In this case, old df-pn suffers from the problem of computing proof and disproof numbers and loops until proof or disproof numbers of nodes that cause infinite loops exceed $\infty - 1$. Old df-pn then gives up solving the node causing infinite loops and tries other branches, eventually resulting in finding a solution.

Figure 4.13 shows node expansions for the problems solved only by df-pn(r). Again, some problems that old df-pn could not solve are easily solved by

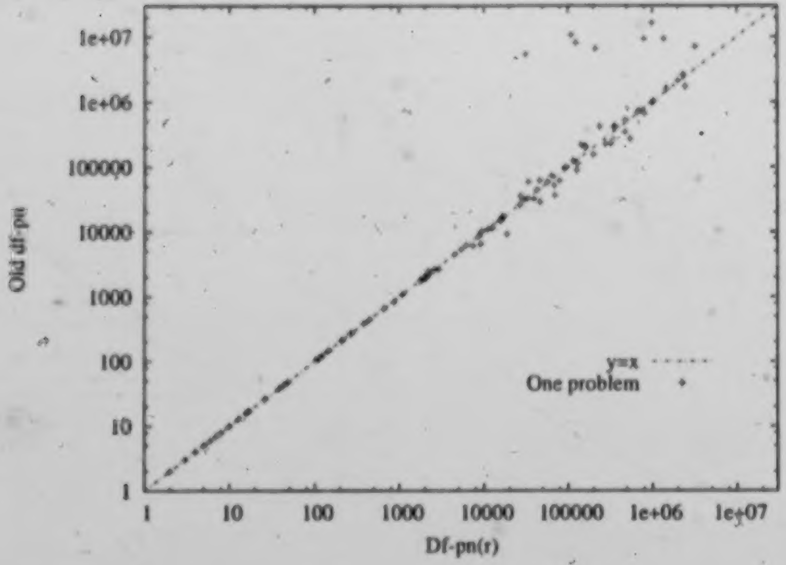


Figure 4.12: Node expansions for problems solved by both versions in checkers.

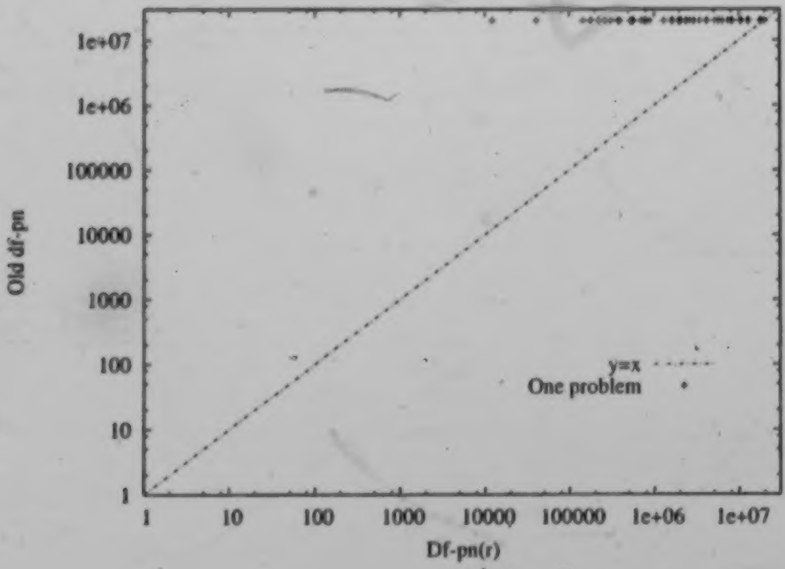
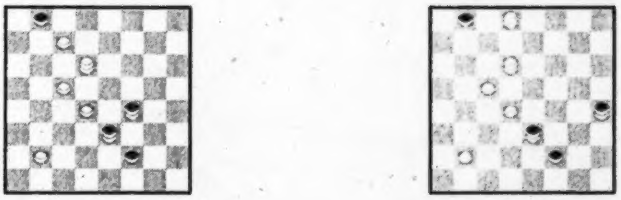
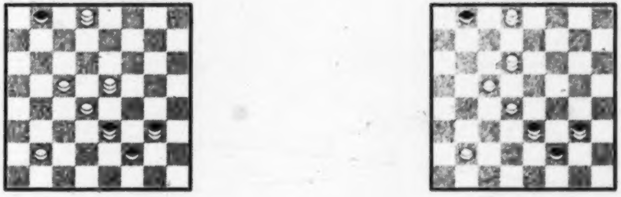


Figure 4.13: Node expansions for problems solved only by df-pn(r) in checkers.



(a) Black to play
The original position
(Position 141)

(b) White to play



(c) White to play

(d) Black to play

Figure 4.14: An example easily solved by df-pn(r) in checkers.

df-pn(r). Figure 4.14(a) shows such a position, which was solved by df-pn(r) in 12,290 nodes, whereas old df-pn could not solve it within 20 million nodes. The problem of computing proof and disproof numbers occurs in the position is confirmed (see Figures 4.14 and 4.15). Algebraic notation is used for moves. The position is a loss for Black. There are two paths to reach the position in Figure 4.14(b) (paths **f4-g5 c7-d8 g5-h4** and **f4-g5 c7-d8 g5-f4 d6-e7 f4-g3 e7-d6 g3-h4**) and (d) (paths **f4-g5 c7-d8 g5-h4 d6-e5 h4-g3 e5-d6** and **f4-g5 c7-d8 g5-f4 d6-e7 f4-g3 e7-d6**). These paths create a cycle, as in Figure 4.15. Old df-pn always exceeds the threshold of the proof number at (c), because the proof number of (d) includes the proof number of (c) and vice versa. On the other hand, df-pn(r) disproves (c) by playing **c5-d6**, which is not played by old df-pn.

4.4 Conclusions

This chapter described a solution to an essential problem of the df-pn algorithm in domains with repetitions. Results in two games, Go and checkers, show

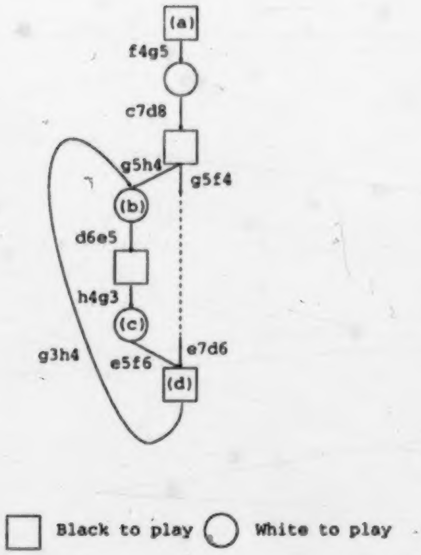


Figure 4.15: An example from checkers showing the problem of computing proof and disproof numbers with repetitions.

that this approach greatly improves the solving abilities of the df-pn solvers without degradation in performance. However, one unanswered question is whether this approach is *complete* or not. In other words, if a problem involving repetitions is given, it is unknown whether df-pn(r) can always solve that problem or not. So far, any problems that are unsolvable because of repetitions have not been found. However, a theoretical analysis of the approach remains as future work.

Chapter 5

Domain Dependent Knowledge for the One-Eye Problem

This chapter gives the details of the one-eye solver. The one-eye solver incorporates the df-pn(r) algorithm. However, since df-pn(r) is not enough to achieve high performance, more effective game-specific methods are added to the solver. After the basic one-eye algorithm is introduced, a few enhancements that safely reduce the search space are described. Then, a method for modeling *ko threats* that affect the outcome of life and death status is given. Experiments show that the one-eye solver with these enhancements is 10 times faster and solves harder problems.

5.1 The Basic One-Eye Algorithm

The basic algorithm, due to Anders Kierulf [32], is quite simple, and has been used as part of the tsume-Go search in the program EXPLORER for many years. It detects single-point eyes and false eyes.

As explained in Section 1.3.2, an instance of the one-eye problem is defined by the defender, attacker, region, crucial stones, and safe attacker stones. The algorithm checks for all points in the region whether they are potential eye points for the defender. Eyes are created by either surrounding empty points or by capturing attacker stones. If a safe eye connected to crucial stones can be created in the region, the defender wins. If no potential eye space remains in the region, the attacker wins. Seki is considered to be a win for the defender.

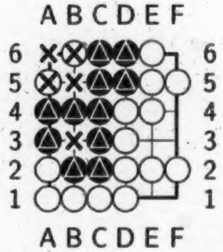


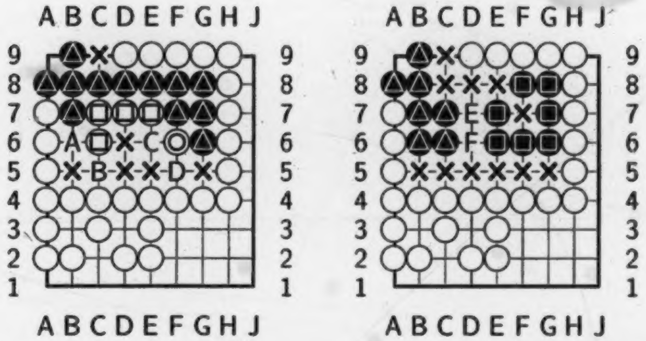
Figure 5.1: An example of seki.

Seki is detected by search. If the defender passes and the attacker cannot win, that position is regard as seki. Figure 5.1, adapted from [53], is an example of seki. White cannot capture the black stones if Black passes.

Whether or not a point E is a *potential eye point* is computed as follows:

- E occupied by unsafe attacker stone: *yes.*
- E occupied by safe attacker stone: *no.*
- E occupied by defender stone: *no.*
- E is empty: check the neighbors and the diagonal neighbors of E.
  - Some direct neighbor is occupied by the attacker: *no.* The safety of the attacker stone does not matter.
  - E is at the edge of the board and at least one diagonal neighbor contains a safe attacker: *no.*
  - At least two diagonal neighbors contain a safe attacker: *no.*
  - Otherwise: *yes.*

A potential eye point is a *safe eye* if all direct neighbors and all but one diagonal neighbor are occupied by defender stones. All diagonal neighbors are needed at the edge of the board. A safe eye is a defender win if the surrounding block is connected to crucial stones, and all crucial stones are connected. The search generates all moves in the region, unless there are forced moves (see Section 5.2.2).



Connection for attacker. Connection for defender.

Figure 5.2: Connections to safe stones.

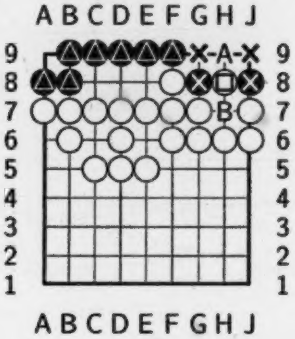


Figure 5.3: Connection to safe stones on protected liberty.

5.2 Game-Specific Search Enhancements

5.2.1 Safety by Connection To Safe Stones

Connectivity is a fundamental aspect of the game of Go. Most Go programs recognize connected blocks. Connections are used to promote unsafe attacker stones to safe, and to prove that an eye is connected to crucial defender stones. Both types of connections help to reduce the search depth.

The current implementation recognizes simple *miai* strategies [51] and some protected liberties for connections. Figure 5.2 gives examples. In the left diagram, White has two ways (**A** and **B**) to connect. The white block marked

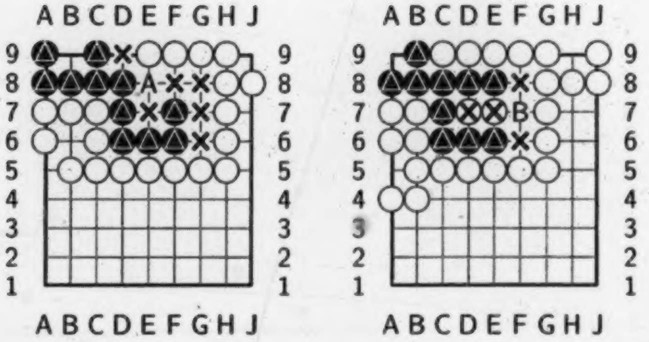


Figure 5.4: Forced moves.

with squares to safe stones, so White can connect even if Black plays first. The stone at **F6** is also safe now, because it has a connection either at *C* or at *D*. Since there is no eye space after recognizing these blocks as safe, this position can be statically evaluated as a loss for Black. Similarly, in the right diagram in Figure 5.2, the connection at **E** or **F** guarantees a win for Black. The algorithm to compute these connections is straightforward. It checks if safe blocks *S* have two liberties to connect to a block *b*. If this is the case, *b* is included in *S* and the two liberties are marked to not be used for other connections. The process continues until no further blocks can be added to *S*.

More safe stones are found by recognizing some forms of protected liberties. Figure 5.3 shows an example. The stone marked with a square has only one connection point at **B** to a safe white block. However, this connection is safe since the stone has another liberty and the opponent cannot play at **B**.

5.2.2 Forced Moves

Forced moves are a safe form of pruning when one player threatens to win immediately. Two kinds of forced moves, *forced attacker moves* and *forced defender moves* are defined.

The first type of forced move is on a point where the defender could complete an eye that is connected to the crucial stones. The left position in Figure 5.4 presents an example. Black can make an eye at **A**. White must play at **A** to stop an immediate win for Black.

The second type of forced move is defined as follows:

1. There is no empty eye space for the defender in the region.
2. There is exactly one unsafe attacker's block b.
3. b has a single-move connection to safe stones. If the defender plays any other move, the attacker can connect b to safety, leaving the defender with no potential eye points.

For instance, in the right position of Figure 5.4 the move at **B** is forced.

Forced moves give a large reduction of the search space by decreasing the branching factor.

5.2.3 Simulation

Kawano's simulation [30] is a quick way to confirm whether a position is proven or not that avoids a normal search. If similar positions for the one-eye problem are plausibly defined, a reduction of the search space can be achieved. In the solver, simulation and dual simulation are applied as follows:

- At an AND node n, assume that one of n's children, n_{child}, is proven at some point in the search. Let n_{child} be proven by the OR player playing m. Let $n_{exception}$ be n's child by playing m at n. Simulation is applied to all unsolved children of n except for $n_{exception}$. $n_{exception}$ is not considered to be similar to n_{child}, because $n_{exception}$ already contains a stone of the AND player at m, where the OR player must play.
- Similarly, at an OR node n, apply dual simulation if one of n's children is disproven.

In Wolf's GoTools, $n_{exception}$ is tried next if one of n's children is (dis)proven [90]. The solver tries nodes except for $n_{exception}$, which is the opposite approach to GoTools. However, this method is believed to be more suitable for the solver using df-pn(r), because of the behavior of df-pn searches. When simulation succeeds, it can make df-pn explore n's children other than n_{child} more deeply

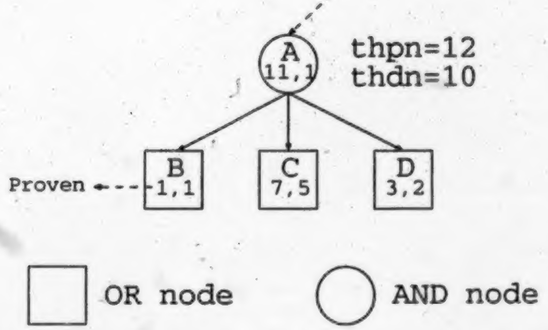


Figure 5.5: An example showing that simulation is effective.

without increasing n's current thresholds. Figure 5.5 shows an example. The proof number is written on the left inside a node and the disproof number on the right in this Figure. Assume that df-pn explores A with $\mathbf{th_{pn}}(A) = 12$, df-pn has just proven B, and D can be proven after $\mathbf{pn}(D)$ is increased to 6. If simulation proves C and df-pn expands D, df-pn explores D with $\mathbf{th_{pn}}(D) = \mathbf{th_{pn}}(A) = 12$. This threshold is enough to prove D. On the other hand, if simulation is not invoked, df-pn searches D with the threshold of the proof number of $\mathbf{th_{pn}}(D) = \mathbf{th_{pn}}(A) - \mathbf{pn}(C) = 12 - 7 = 5$. In this case, the threshold of the proof number of 5 is not enough to prove D. Df-pn must reach A again in later iterations with larger thresholds, which usually needs more node expansions.

$n_{exception}$ contains a different proof or disproof tree from n_{child}. On the other hand, the other children of n can be similarly proven by n_{child}'s proof tree. Therefore, proving children other than $n_{exception}$ first has more chances to assign a larger threshold for proving $n_{exception}$ without increasing n's threshold.

This use of simulation is much more extensive than in tsume-shogi [30]. See Section 5.4.4 for a further discussion.

5.2.4 Heuristic Initialization

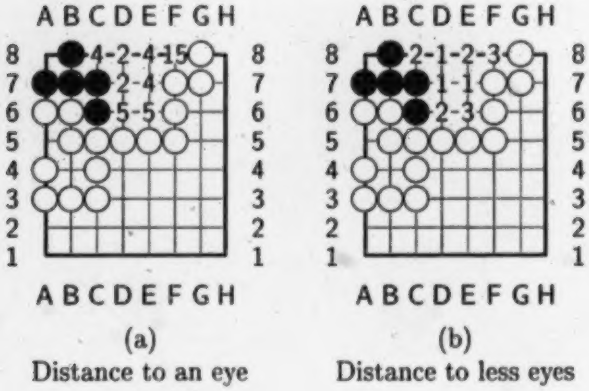
If the basic df-pn algorithm encounters a leaf node, the proof and disproof numbers for that node are initialized to 1. One possibility to further enhance performance is to use heuristic initialization of proof and disproof numbers as in Nagai's df-pn$^+$ [57], or in [2, 10]. The current implementation uses problem-specific evaluation functions that were originally implemented in EXPLORER.

Let the defender be the player to disprove, and the attacker be one to prove. A heuristic estimate of the "distance" to make an eye is computed to initialize a disproof number for each move on a point p. This initialization function for disproof numbers first checks potential eye points that are adjacent or diagonally adjacent to p. The function counts how many moves in a row the defender must make to create a safe eye. The initialization function assigns the minimum value among these numbers to p. Figure 5.6(a) illustrates an example. Each number represents a disproof number. Potential eye points exist at **C8**, **D8**, and **D7**. To create an eye on **C8** requires two black stones at **D7** and **D8**, whereas for **D7** five black stones are needed: **D6**, **D8**, **E7**, and two of the three diagonally adjacent points **C8**, **E6**, and **E8**. The disproof number of a black move on **D8** is 2, the minimum distance to an eye that this move contributes to. Higher disproof numbers are assigned to moves that do not directly help the defender make an eye. In the current implementation, the highest value among the moves that directly help to make an eye plus 10 points is assigned to such an indirect eye-making move. For example, the 15 points for **F8** includes a 10 point penalty and the highest non-penalized value in this position is 5 (i.e., either **D6** or **E6**). Note that moves such as **F8** may still help to make eyes indirectly and therefore cannot be pruned. Figure 5.7 shows an example in which the only winning move does not decrease the distance to an eye as measured by this heuristic. **G9** is a move that is not played next to an eye space, but is necessary to connect to an eye.

Similarly, a heuristic distance to make the defender eye-less is computed to initialize proof numbers. Let ne be the number of eye spaces for the current position, and $b(m)$ be the number of eye spaces that move m directly breaks. Then the heuristic distance $h(m)$ is computed by:

$$h(m) = \max(1, ne - b(m)).$$

$b(m)$ checks whether m directly connects to the safe stones, or two miai



(a)
Distance to an eye

(b)
Distance to less eyes

Figure 5.6: Heuristic initialization for the one-eye problem.

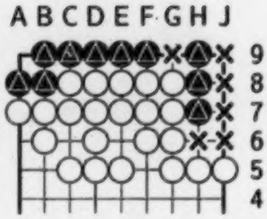


Figure 5.7: Example of an indirect move that must be generated.

points to connect m to safe attacker stones exist. Then, $b(m)$ returns the number of m's direct and diagonal eye points that m can break.

Figure 5.6(b) illustrates an example. Again, there are three potential eye points. Since White's move **D8** breaks potential eye points at **C8** and **D7**, the initialization function returns $3 - 2 = 1$ as the proof number for **D8**. The proof number to **D7** can be set to 0, since it breaks all potential eye points. However, since the current implementation does not always correctly estimate $b(m)$, the proof number of each move is initialized to at least 1. The current implementation for $b(m)$ assumes that two miai points for m are always available to establish m's connection. However, these points may have already been used when promoting another unsafe attacker block is promoted to be safe.

When proof and disproof numbers are initialized at leaf nodes, the reexpansion overhead at interior nodes increases in the implementation. Let n be an OR node, n_c be n's child selected by df-pn, and pn_2 be the second largest proof number among n's children. The standard df-pn algorithm computes the threshold of the proof number at n as follows:

$$\mathbf{th}_{\mathbf{pn}}(n_c) = \mathbf{min}(\mathbf{th}_{\mathbf{pn}}(n), pn_2 + 1).$$

Let the value added to pn_2 be the *unit* of the proof number. Unit = 1 is a small value for initialized proof numbers. For example, assume that n has two leaf nodes and $\mathbf{th}_{\mathbf{pn}}(n) = \infty - 1$. Since the standard df-pn algorithm sets the proof numbers of the leaf nodes to 1, $\mathbf{th}_{\mathbf{pn}}(n_c) = 2$. Df-pn explores n_c until $\mathbf{pn}(n_c) = 2$, which doubles the proof number of n_c. However, if df-pn with heuristic initialization initializes the proof numbers of these two leaf nodes to 6 and the unit is 1, it sets $\mathbf{th}_{\mathbf{pn}}(n_c) = 7$, which is only a factor of $\frac{1}{6}$ larger than the proof number at the leaf nodes. Hence, the termination condition of df-pn can be satisfied more frequently with the heuristic initialization. This results in more reexpanded interior nodes.

To reduce the number of reexpansions, the modified df-pn(r) algorithm tries to assign a larger unit than 1. The proposed scheme tends to have a larger unit if larger proof and disproof numbers are assigned to leaf nodes.

When computing the proof number at an OR node n, the evaluation function that initializes proof numbers is called to compute the heuristic scores of all moves and computes the average of these scores pn_{ave}. The threshold of the proof number at n_c is adjusted as follows:

$$\mathbf{th_{pn}}(n_c) = \mathbf{min}(\mathbf{th_{pn}}(n), pn_2 + pn_{ave}).$$

If n is an AND node, the threshold of the disproof number for n_c is set analogously. The same scheme as in the standard df-pn algorithm is used for the other cases. Figure 5.8 presents pseudo-code of nonuniform threshold increments. $\Delta Eval$ is an evaluation function that returns an initialized (dis)proof number for n. NumberOfMoves returns the number of legal moves for n.

5.3 Ko and Ko Threats

Sometimes the outcome of a one-eye problem depends on ko. It is therefore important to model ko threats and ko recaptures in the search algorithm.

The approach taken in GoTools can require several searches [88]. The parameter to each search is how many ko recaptures are allowed for a specified ko winner.

The current implementation allows only two options: one is to forbid any immediate ko recaptures; the other is to always allow ko recaptures for a designated ko winner. It searches in one or two phases. The first search of a position, phase 1, forbids immediate ko recaptures, but marks nodes where such moves exist. If the search result depends on marked nodes, in phase 2 a re-search is performed. The loser of the phase 1 search becomes the designated ko winner for phase 2. Even if there are at least two ko in the region, this approach works. The ko winner cannot immediately recapture both ko at once, and can win only one ko.

Phase 2 reuses the contents of the transposition table from phase 1. The following implementation of the transposition table aims at reducing the amount of re-search:

```
// Iterative deepening at each node
void MID(node &n) {
  ...
  // Iterative deepening
  while ($n.\phi > \Delta\mathrm{Min}(n)$  &&  $n.\delta > \Phi\mathrm{Sum}(n)$)  {
    $n_c$ = SelectChild($n, \phi_c, \delta_2$);
    // Update thresholds
    $n_c.\phi = n.\delta + \phi_c - \Phi\mathrm{Sum}(n)$;
    $n_c.\delta = \min(n.\phi, \delta_2 + \Delta\mathrm{Average}(n))$;
    MID($n_c$);
  }
  ...
}

// Compute average $\delta$ of $n$'s children that are heuristically initialized
int ΔAverage(node $n$) {
  int $ave$ = 0;
  for (each child $n_{child}$ of $n$) {
    $ave = ave + \Delta\mathrm{Eval}(n_{child})$;
  }
  return ($ave$ / NumberOfMoves($n$));
}
```

Figure 5.8: Pseudo-code of the df-pn algorithm with nonuniform threshold increments.

1. The Zobrist hash function [92] is modified to account for a stone captured for ko in the previous move, to differentiate identical positions with different histories.

2. Two flags, one for each color, in each transposition table entry keep track of any possible ko captures in the subtree below that node. If there is a ko capture for a player, the flag for the other player is set to indicate that a ko recapture will be allowed after that node in a re-search. When a node n is proven (similarly for disproven), flags are set as follows:
   - If n is an OR node and n_c is n's proven child, n's flags are set the same as n_c's flags.
   - If n is an AND node, the flag is set if and only if the flag of at least one of the children is set.

In the phase 2 re-search, many phase 1 (dis)proofs can be reused. For example, assume that a node is proven and the flag for the ko winner is not set. Then the proof from the transposition table can be used. Similarly, disproofs can also be reused. Even for nodes that are not proven or disproven, the proof and disproof numbers from phase 1 are valuable information for directing the re-search.

Re-searches usually have a low overhead, since the previous results are kept in the transposition table and the table entries are reused in most cases. However, if the solution changes dramatically by ko compared to the solution from the first search, a larger amount of new search is necessary (see an example in Section 5.4).

5.4 Empirical Results

5.4.1 Setup of Experiments

The following abbreviations are used for the methods and enhancements described above:

- **Df-pn(r)**: The modified df-pn algorithm.

Table 5.1: Performance for successively switching on enhancements.

Enhancements used	Number of problems solved	Total time (sec) 138 problems	Total nodes expanded 138 problems	Nodes expanded per second
(1): **Df-pn(r)**	138	1,468	64,532,221	43,947
(2): (1) + **AC**	143	506	20,807,755	41,095
(3): (2) + **DC**	144	587	22,109,250	37,653
(4): (3) + **FDM**	144	398	15,228,689	38,242
(5): (4) + **FAM**	146	267	10,508,493	39,257
(6): (5) + **SIM**	152	170	7,541,088	44,128
(7): (6) + **EYEDIST**	151	140	6,177,114	43,977
(8): (7) + **FEYEDIST**	157	75	3,888,457	51,777
Total problems	162	-	-	-

- **AC**: Connections to safe stones for attacker.
- **DC**: Connections to crucial stones for defender.
- **FAM**: Forced attacker's moves.
- **FDM**: Forced defender's moves.
- **SIM**: Simulation and dual simulation.
- **EYEDIST**: Heuristic initialization for the defender to make an eye.
- **FEYEDIST**: Heuristic initialization for the attacker to prevent an eye.

5.4.2 Adding Enhancements

Table 5.1 shows the results on the test set in Section 3.5.1, starting with df-pn(r) and switching on enhancements one by one. The total execution time and number of nodes expanded were computed using the subset of 138 problems that are solved by all methods (1) - (8) in the table. All problems solved by (1) are solved by (2) - (8). Moreover, except for one case between (6) and (7), the version with a new enhancement solved all problems solved by

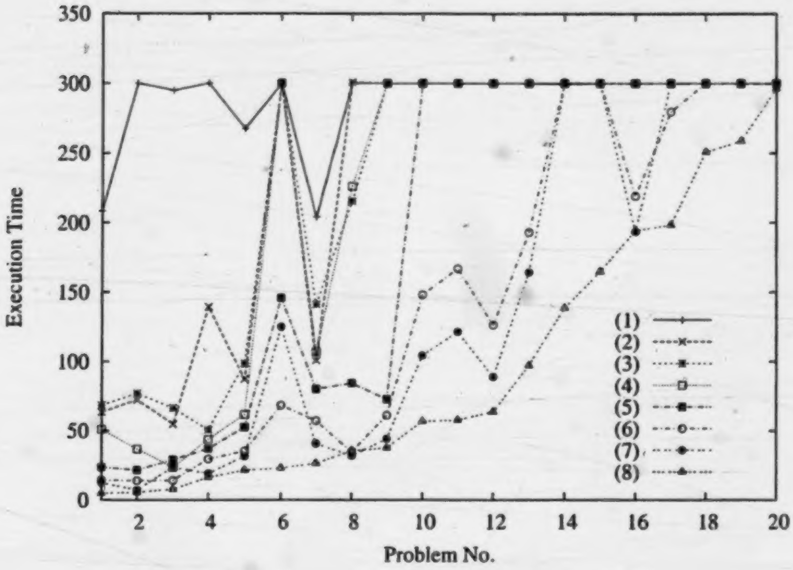


Figure 5.9: Performance for 20 hard problems for each enhancement.

the versions before. One problem was solved close to the time limit by version (6), but was unsolved within 5 minutes by version (7).

Search speed decreases a little with more enhancements, but improves again with simulation. As is explained in Chapter 3, simulation provides a fast way to generate moves, faster than the current normal move generator, which has some overhead such as checking connections and looking up the transposition table information on children. Since computing the distance to an eye or no eye is simple, the search speed becomes dependent on the frequency of simulation calls.

Figure 5.9 plots the time for the 20 hardest problems for all versions, successively switching on all enhancements. Enhancements are successively turned on as in Table 5.1. A time of 300 seconds means that the problem was not solved. This vividly demonstrates the importance of adding knowledge-based enhancements. For example, problem oneeyee.10.sgf with White to play, could not be solved by (1) in 300 seconds, whereas version (8) with all enhancements solved it in 5.6 seconds with 306,756 nodes.

Table 5.2: Performance for turning off single enhancement.

Enhancements used	Number of problems solved	Total time (sec) 148 problems	Total nodes expanded 148 problems	Nodes expanded per second
(1): **AC**	150	1,007	54,382,555	54,004
(2): **DC**	156	464	26,447,241	56,946
(3): **FDM**	154	622	30,996,740	49,799
(4): **FAM**	153	646	32,414,442	50,173
(5): **SIM**	152	889	36,421,076	40,968
(6): **EYEDIST**	154	665	33,672,519	50,648
(7): **FEYEDIST**	151	796	35,611,877	44,763
(8): All turned on	157	517	27,310,377	52,799
Total problems	162	-	-	-

5.4.3 Leaving out Enhancements

Two experiments measure whether enhancements are effective in isolation, and how much they contribute overall. In Table 5.2, results for switching off a single enhancement is shown. Table 5.3 shows results for using only a single enhancement. The results show that all enhancements except for **DC** improve performance of the solver. Adding **DC** shows both advantages and disadvantages.

5.4.4 Performance of Simulation

Table 5.4 shows the performance of simulation in phase 1 searches. Since the method is applied in a very basic way, 52% success seems to be a good initial result, with plenty of room for further refinements. Such refinements will be one of the possible future research topics.

5.4.5 Re-searches for Ko

Table 5.5 shows a summary of the overhead incurred by re-searches for ko. In phase 1, immediate ko recaptures are not allowed. Phase 2 are the re-searches with a designated ko winner. The results in this table are also with

Table 5.3: Performance for turning on single enhancement.

Enhancements used	Number of problems solved	Total time (sec) 135 problems	Total nodes expanded 135 problems	Nodes expanded per second
(1): **Df-pn(r)**	138	702	32,238,410	45,936
(2): **AC**	143	263	10,605,909	40,261
(3): **DC**	137	776	31,396,386	40,439
(4): **FDM**	139	696	32,238,410	46,292
(5): **FAM**	141	392	18,918,746	48,239
(6): **SIM**	146	308	15,484,676	50,170
(7): **EYEDIST**	141	420	19,489,233	46,379
(8): **FEYEDIST**	142	379	19,139,981	50,441
Total problems	162	-	-	-

Table 5.4: Performance of simulation for all 157 solved problems (all enhancements on, phase 1 searches only).

Total nodes	Nodes by **SIM**	**SIM** calls	Successful calls
74,841,218	28,450,185 (38.0%)	6,808,439	3,570,662 (52.4%)

Table 5.5: Overhead for ko re-searches.

Total nodes (157 problems)	
Phase 1	Phase 2
74,841,218 (89.0%)	9,243,534 (11.0%)

all enhancements.

The overhead is small, but of course this is mainly a property of the test set used, which contains only a few cases with complex ko fights. In the worst case encountered, problem oneeyed.3.sgf with Black to play, phase 1 took 178 nodes and phase 2 took 787 nodes.

5.4.6 Reexpansion of Interior Nodes

The performance of the version that sets larger thresholds (UNIT-N) against the version that does not set larger thresholds (UNIT-1), described in Section 5.2.4 was compared. Since UNIT-1 needed more time to solve the same subset of test positions, Both versions were run for 400 seconds.

Table 5.6 summarizes the performance. The number of nodes explored by simulation is excluded to compute the ratio of reexpanded nodes. UNIT-1 suffers from a larger overhead to reexpand interior nodes than UNIT-N, resulting in 28% longer total execution time and 22% extra nodes. In Figure 5.10 the node expansions of UNIT-N are plotted on the X-axis against UNIT-1 on the Y-axis on logarithmic scales. A point above the diagonal means that UNIT-N performed better. With small-size problems, it is hard to see a difference between two versions. However, with larger problems, UNIT-N explores slightly less nodes than UNIT-1. Therefore, UNIT-N is believed to scale better.

In Seo's experiments in shogi, the ratio of reexpansions was about 20% [77]. Since information achieved dynamically is usually more reliable than static evaluation, the 32% is still a very small price to pay to achieve more cut-offs. However, investigating the trade-off between the ratio of reexpansions and decreasing the total execution time remains as future work.

Table 5.6: Performance comparison between UNIT-N and UNIT-1.

Methods used	Number of problems solved	Total time (sec) (157 problems)	Total nodes expanded (157 problems)	Ratio of reexpanded nodes
UNIT-1	157	2,518	102,515,362	44.7%
UNIT-N	157	1,975	84,084,752	32.0%
Total problems	162	-	-	-

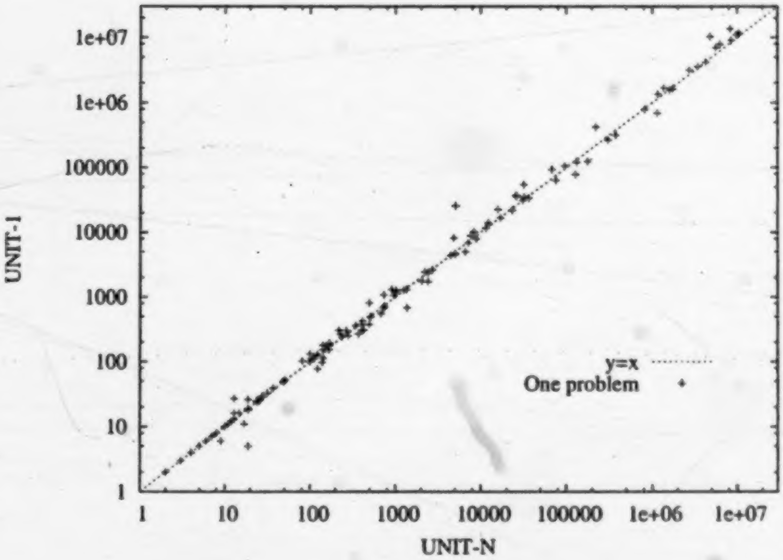


Figure 5.10: Comparison of node expansions between UNIT-N and UNIT-1.

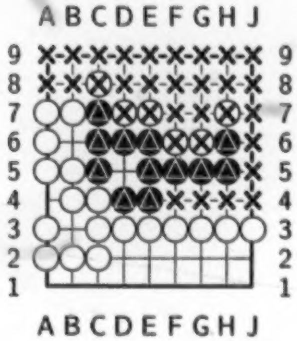


Figure 5.11: Black to play and live: A currently unsolved problem.

Table 5.7: List of unsolved problems.

Problem name	Color to play
oneeyed.7.sgf	Black
oneeyed.9.sgf	Black
oneeyed.9.sgf	White
oneeyee.2.sgf	Black
oneeyee.11.sgf	White

5.4.7 Currently Unsolved Problems

The solver currently cannot solve the 5 problems in the test suite listed in Table 5.7. Figure 5.11 shows an example. All unsolved problems feature large regions with many possible moves. Besides, some problems such as Figure 5.11 stretch the limits of the one-eye problem, and should rather be treated as semeai or tsume-Go problems. In Figure 5.11, for example, making one eye for Black depends on whether white stones adjacent to black crucial stones can make two eyes or not.

5.4.8 Comparison to Other Solvers

As far as I know, this is the first program specialized for solving one-eye problems. Since the algorithm solves only one-eye problems, it is hard to make a fair comparison with general tsume-Go solvers. Evaluation for two eyes is much harder than for one eye, and many years of hard work have gone into the development of the Go knowledge in programs such as GoTools. the comparison on the tsume-Go solver will be left to the next chapter.

5.5 Conclusions and Future Work

The results of the work on applying df-pn(r) to the one-eye problem in Go are very encouraging. There are numerous possible enhancements, both for improving the search algorithm and for adding Go-specific knowledge. Examples are recognizing larger eyes, refining the knowledge about connections, generalizing forced moves, and search in open-ended areas.

To apply these ideas to other problems in Go is also an interesting research topic. Examples include full tsume-Go (two-eye problems), tactical capture search and connection search. Applying the ideas to tsume-Go is the topic of the next chapter.

Chapter 6

Domain Dependent Knowledge for the Tsume-Go Problem

This chapter deals with techniques for the tsume-Go solver, called TsumeGo Explorer. The ideas invented for the one-eye problem are applicable to tsume-Go with slight modifications. Experiments show that the approach in the chapter is very promising. TsumeGo Explorer outperforms GoTools, the previously best tsume-Go solving program.

6.1 The Basic Two-Eye Algorithm

The basic two-eye algorithm is similar to the basic one-eye algorithm. At first, all potential eye points are computed. However, finding two safe eyes using the basic-eye algorithm is not enough, since it does not work when two eyes are diagonally adjacent as in Figure 6.1. In this example, although neither E_1 nor E_2 is a safe eye for the algorithm of Chapter 5, the black stones are alive. The notion of a ***safe eye E_1 with respect to point E_2*** is introduced as follows:

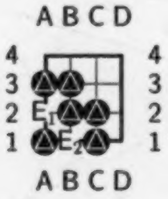


Figure 6.1: Example in which the basic one-eye algorithm fails.

- E_2 must not be E_1's direct neighbor.
- If E_1 is at the edge of the board, all diagonal neighbors except for E_2 are occupied by defender stones.
- If E_1 is not at the edge of the board, all direct neighbors and all but one diagonal neighbor excluding E_2 are occupied by defender stones.

Two potential eye points E_1 and E_2 are *two safe eyes* if the following conditions hold:

- E_1 is a safe eye with respect to E_2.
- E_2 is a safe eye with respect to E_1.
- E_1 and E_2 share all defender blocks.

In the basic two-eye algorithm, if the defender can construct two safe eyes in a given region, the defender wins. A group with two safe eyes is always safe, since all blocks constructing two eyes always have two liberties E_1 and E_2, which can not be played at a time by the attacker. Hence, this is a special case of Benson's unconditional safety [6]. If less than two nonadjacent potential eye points remain in the region, the attacker wins. As defined in Section 5.1, seki is detected by search and is a win for the defender.

Currently, this approach cannot detect live stones that do not have two safe eyes. Figure 6.2 illustrates an example of such a position. In this example, black stones marked with triangles are alive, although they merely have two false eyes. In the above algorithm, this position is dead, because it is considered to have no eye space. This case is currently ignored, because it occurs very rarely. Such a case could be detected by implementing Benson's algorithm [6] or Müller's method for static detection of safe stones [51].

One improvement that is currently added only in the tsume-Go solver is detection of large eyes. If two adjacent potential eye points are surrounded by defender stones, they are considered to be a large eye. Let E_1 be a potential eye point. A *large safe eye* E_1 *with respect to point* E_2 is defined as follows:

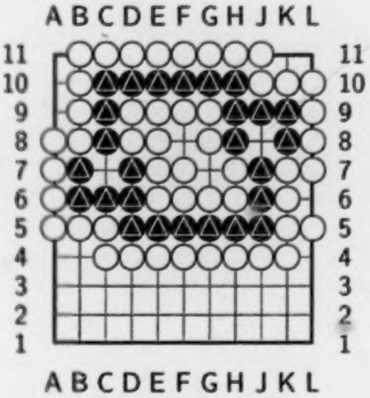


Figure 6.2: A live position without two eyes.

- E_2 must be empty.
- Check all adjacent points to E_1. If there is only one point E_3 that is either empty or an unsafe attacker stone, the algorithm checks the next process. Otherwise, E_1 is not a large safe eye.
- Neither E_1 nor E_3 is adjacent to E_2.
- All direct neighbors of E_1 and E_3 must be occupied by defender stones.
- Let E_p be E_1 or E_3. Check the following conditions for both E_1 and E_3;
  - If E_p is at the edge of the board, all diagonal neighbors of E_p except for E_2 are occupied by defender stones.
  - If E_p is not at the edge of the board, all direct neighbors and all but one diagonal neighbor excluding E_2 are occupied by defender stones. If one diagonal neighbor except for E_2 does not contain a defender stone for E_p, all diagonal neighbors except for E_2 must be occupied by defender stones for the other case.

Because E_2 must be empty in the above definition, it excludes the case in which stones are captured by playing inside the eye space. Figure 6.3 shows an example. In this example, A is a large eye point with respect to B, but B

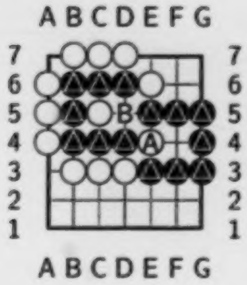


Figure 6.3: An example that is not a large safe eye.

is not a large eye point with respect to A. If the defender can construct either two large safe eyes, one large safe eye and one safe eye, or two safe eyes, the defender wins. These two eyes are also safe, because they are a special case of Benson's unconditional safety, or the existence of moves that lead to Benson's safety by alternating play is guaranteed. This is proven as follows:

- Since two safe single point eyes are safe, the cases of two large safe eyes, and one large safe eye and one safe eye are considered.
- The state in which there are two stones of the attacker in a large eye is impossible.
- If there is a stone of the attacker in the large eye, the fact that the defender has another safe eye or one safe eye guarantees that the attacker cannot play inside the large eye. Hence, the defender block have two sure liberties.
- If there are two empty points in a large eye with respect to point p and the attacker plays inside, the defender can capture the stone of the attacker by filling the large eye, making the large eye a safe single point eye with respect to p.

6.2 Game-Specific Knowledge

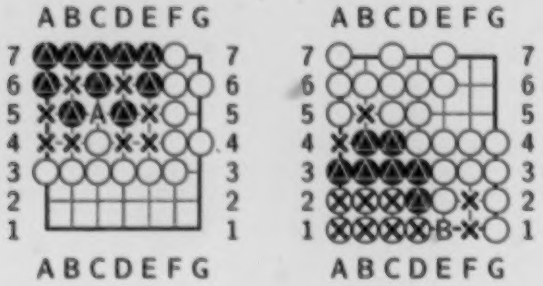
Techniques invented for the one-eye solver are incorporated into the tsume-Go solver. Each technique is briefly reviewed:

Safety by connections to safe stones Connections that promote unsafe attacker stones to safe are considered. However, connections that help the defender make two eyes are not implemented yet, because of the following complexities:

- There are more cases to consider in tsume-Go than in the one-eye problem, such as:
  - One region containing two complete eyes is recognized and connected to a crucial block.
  - Two single eyes are recognized and connected. Then, the eyes are connected to an crucial block.
- In the definition of the one-eye problem, all crucial stones must be connected to an eye. The algorithm to compute connections is invoked once to check if all of them are connected. On the other hand, following Wolf's definition for tsume-Go, the defender wins if any one of the crucial stones is alive. Some crucial stones can possibly be captured. If there are n crucial blocks, connections must be computed to check if one of the blocks are proven to have two safe eyes. Therefore, the algorithm computing connections must be invoked n times, which may incur a higher overhead to compute.

Forced Moves Forced moves similar to the one-eye solver are incorporated into the tsume-Go solver. A *forced attacker move* is generated on a point where the defender could make two eyes such as point **A** in Figure 6.4(a). A *forced defender move* is generated on a point where the defender must play, checking the following conditions:

- There is only one unsafe attacker block b which has a single-move connection to safe stones.
- If the defender plays any other move and the attacker connects b to safety, the defender has either no potential eye point, one potential eye point, or two adjacent potential eye points.



(a) (b)

Figure 6.4: Forced moves.

In this case, the defender is forced to play so that the attacker cannot connect *b* to safe stones. Otherwise, the attacker could steal the potential eye points that are necessary for the defender by playing on that point. An example is point **B** in Figure 6.4(b).

Simulation Simulation is invoked in the same way as in the one-eye solver.

Heuristic Initialization Assume that the attacker is the player to prove a position, and the defender is one to disprove. In order to initialize proof numbers, the same evaluation function that the one-eye solver uses is incorporated into the tsume-Go solver. The single-player distance to make two eyes is computed to initialize disproof numbers. This is similar to the heuristic function from the old EXPLORER tsume-Go solver [32]. The heuristic function tries to approximate the current "distance" to create two eyes on two potential eye points E_a and E_b. Then, based on the current distance, the distance to two eyes after move m is played is computed, which is used as an initial disproof number for m. This process can be summarized as follows:

1. For each potential eye point E, *DistOne*(E), the number of consecutive defender moves to make an eye at E, is computed as explained in Section 5.2.4.

2. Let E_a and E_b be potential eye points. The distance to two eyes is

approximated by *DistOne*, defined as follows:

$$DistTwo(E_a, E_b) = DistOne(E_a) + DistOne(E_b).$$

3. Assume that E_1 is a potential eye point that is a direct or diagonal neighbor of move m. Let E_2 be a potential eye point that contains at least one common defender stone to make a safe eye at E_1 with respect to E_2. For all combinations of such E_1 and E_2, the smallest value of $Eval(E_1, E_2, m)$ is computed to set as an initialized disproof number. $Eval(E_1, E_2, m)$ is defined as follows:

   - If m helps to make one eye in both E_1 and E_2, for example if m is a direct neighbor of both E_1 and E_2, $Eval(E_1, E_2, m)$ is computed as follows:

     $$Eval(E_1, E_2, m) = max(1, DistTwo(E_1, E_2) - 2).$$

     Note that 2 is subtracted from $DistTwo(E_1, E_2)$, since $DistOne(E_1)$ and $DistOne(E_2)$ are used to measure $DistTwo(E_1, E_2)$, and m helps to decrease the distance to make an eye on both E_1 and E_2.

   - Otherwise, $Eval(E_1, E_2, m) = max(1, DistTwo(E_1, E_2) - 1)$.

4. If E_1 and E_2 are not found, move m does not help decrease the distance to two eyes. In the implementation, 10 points plus the largest score of the move which directly helps to make two eyes is assigned to m as a heuristic disproof number. So far there is no distinction between the moves that directly helps to make only an eye and the ones that decrease the distance neither to one eye nor to two eyes. This is a possible future extension to this method.

Ko and Ko Threat The treatment is identical to the one-eye solver. If ko is involved in a proof or disproof tree in the first phase, a re-search is performed by assuming that the loser can immediately re-capture ko as often as needed.

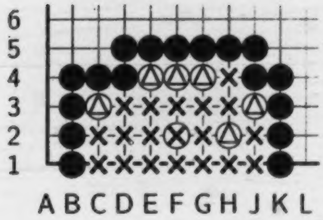


Figure 6.5: A position in Wolf's test collection.

6.3 Experimental Results

6.3.1 Setup

Two kinds of test suites were prepared for experiments:

- A subset of 40,000 tsume-Go problems generated by GoTools [85] was chosen. In the problem collection, there are 6 volumes. Each volume contains 14 levels. All 283 positions in the hardest category in volume 6, called *LV6.14*, are chosen. Figure 6.5 shows an example. The problems were tried for either color playing first, resulting in a total of 566 instances.

- Since the one-eye problems used for experiments in the previous chapter can be seen as tsume-Go problems, these problems were used, to compare the performance against GoTools. This test suite is called *ONEEYE*.

As in the last chapter, the following abbreviations are used for the methods and enhancements described above:

- **Df-pn(r)**: The modified df-pn algorithm.

- **2POINTEYE**: Recognition of two point eyes.

- **AC**: Connections to safe stones for attacker.

- **FAM**: Forced attacker's moves.

- **FDM**: Forced defender's moves.

Table 6.1: Performance for successively switching on enhancements in LV6.14.

Enhancements used		Number of problems solved	Total time (sec) 564 problems	Total nodes expanded 564 problems	Nodes expanded per second
(1):	**Df-pn(r)**	564	6,262	399,195,987	63,743
(2):	(1) + **2POINTEYE**	565	5,340	332,933,309	62,335
(3):	(2) + **AC**	564	3,828	185,639,307	48,488
(4):	(3) + **FDM**	564	3,061	136,639,362	44,629
(5):	(4) + **FAM**	566	1,480	63,049,713	42,572
(6):	(5) + **SIM**	566	1,146	54,265,265	47,344
(7):	(6) + **EYEDIST**	566	994	43,076,112	43,323
(8):	(7) + **FEYEDIST**	566	808	36,592,350	45,287
Total problems		566	-	-	-

- **SIM**: Simulation and dual simulation.
- **EYEDIST**: Heuristic initialization for the defender to make two eyes.
- **FEYEDIST**: Heuristic initialization for the attacker to prevent two eyes.

6.3.2 Performance on Enhancements

Table 6.1 shows the results in LV6.14, starting with df-pn(r) and switching on enhancements one by one. The total execution time and number of nodes expanded were computed using the subset of 564 problems that are solved by all methods (1) - (8) in the table. All problems solved by (1) are solved by (2) - (8). Besides, each version with a new enhancement solved all previously solved problems except two cases between (2) and (3), and between (2) and (4). One problem was solved close to the time limit by version (2), but was unsolved within 5 minutes by version (3) and (4). Table 6.2 shows results on removing one enhancement. Table 6.3 shows results on adding single enhancement. These experiments mostly confirm the results of Chapter 5 for the one-eye problem. However, there are two main differences:

- The ratio of success for simulation is lower than in Chapter 5. This can

Table 6.2: Performance for turning off single enhancement in LV6.14.

Enhancements used		Number of problems solved	Total time (sec) 566 problems	Total nodes expanded 566 problems	Nodes expanded per second
(1):	**2POINTEYE**	566	1,048	47,781,611	45,581
(2):	**AC**	566	1,551	92,971,700	59,955
(3):	**FDM**	566	1,154	53,000,753	45,929
(4):	**FAM**	566	1,372	66,147,069	48,213
(5):	**SIM**	566	1,314	50,478,657	38,416
(6):	**EYEDIST**	566	1,089	52,745,197	48,435
(7):	**FEYEDIST**	566	1,167	50,228,127	43,051
(8):	All Turned on	566	933	42,192,836	45,222
	Total problems	566	-	-	-

Table 6.3: Performance for turning on single enhancement in LV6.14.

Enhancements used		Number of problems solved	Total time (sec) 562 problems	Total nodes expanded 562 problems	Nodes expanded per second
(1):	**Df-pn(r)**	564	5,721	373,314,568	65,249
(2):	**2POINTEYE**	565	4,815	308,313,657	64,037
(3):	**AC**	564	4,087	213,151,669	52,151
(4):	**FDM**	563	5,951	373,278,106	62,723
(5):	**FAM**	564	3,373	201,512,000	59,740
(6):	**SIM**	564	3,409	237,915,070	69,797
(7):	**EYEDIST**	565	5,165	273,697,298	52,988
(8):	**FEYEDIST**	565	4,035	274,566,759	68,048
	Total problems	566	-	-	-

Table 6.4: Performance data on simulation for all 566 solved problems in LV6.14. All enhancements on. Phase 1 searches only.

Total nodes	Nodes by **SIM**	**SIM** calls	Successful calls
24,940,863	9,006,439 (36.1%)	3,009,733	1,274,218 (42.3%)

Table 6.5: Overhead for ko re-searches.

Total nodes (566 problems)	
Phase 1	Phase 2
24,940,863 (59.1%)	17,251,973 (40.9%)

be explained by the greater complexity of tsume-Go compared to the one-eye problem. For example, when all enhancements are turned on, the success rate of simulation was 42% (see Table 6.4), compared to 52% in Chapter 5 for the one-eye problem.

- TSUMEGO EXPLORER performed a much larger amount of re-search for ko in LV6.14 (see Table 6.5). 40.9% of all node expansions were needed for re-search for ko, whereas the number in Chapter 5 was 11.0%. Since LV6.14 contained a lot of problems that require a number of ko threats to win, the solver suffered from such high overhead. Similar behavior occurs in GoTools if many ko are involved in the problem. Wolf mentions in [87] that for a complete search all ko's with threats and answers must be played out and the tree typically deepens by a factor of 1.5 in the presence of ko's.

6.3.3 Comparison with GoTools

TSUMEGO EXPLORER is compared with a general tsume-Go solver to assess its performance. GoTools was chosen, since it has been considered to be the best tsume-Go solver for 15 years. GoTools was run under the same experimental conditions.[1] Since GoTools and TSUMEGO EXPLORER use different schemes for dealing with ko and ko threats, the problems whose results depend on the number of ko threats available were excluded. As a result, 418 positions in LV6.14 and 148 positions in ONEEYE were used.

Tables 6.6 and 6.7 compare two solvers on LV6.14 and ONEEYE. These indicate that TSUMEGO EXPLORER surpasses the solving ability of GoTools. Although both programs solve all problems in LV6.14, TSUMEGO EXPLORER

---

[1] Sometimes GoTools had a problem on enclosing positions. In this case, the outside of regions was filled out by attacker stones to make GoTools correctly enclose them.

Table 6.6: Performance comparison between TSUMEGO EXPLORER and GoTools in LV6.14.

	Number of problems solved	Total time (sec) (418 problems)
GoTools	418	1,235
TSUMEGO EXPLORER	418	448
Total problems	418	-

Table 6.7: Performance comparison between TSUMEGO EXPLORER and GoTools in ONEEYE.

	Number of problems solved	Total time (sec) (119 problems)
GoTools	119	957
TSUMEGO EXPLORER	142	47
Total problems	148	-

solves them about 2.8 times faster than GoTools. In ONEEYE, all problems solved by GoTools were also solved by TSUMEGO EXPLORER. In addition, TSUMEGO EXPLORER solved these problems more than 20 times quicker than GoTools. All results are obtained with the original version of GoTools provided by Wolf.

Figures 6.6 plots the execution time for problems solved by both programs. The time spent by TSUMEGO EXPLORER is plotted on the X-axis against GoTools on the Y-axis on logarithmic scales. In points above the diagonal TSUMEGO EXPLORER performed better. In case of easy problems, it is hard to see a difference in performance between the two solvers. Sometimes GoTools performs better than TSUMEGO EXPLORER, because of its large amount of hand-coded knowledge. For example, GoTools solved the position in Figure 6.7 in 0.08 seconds. The search tree of GoTools contained 167 leaf nodes. The deepest depth explored by GoTools is 19. GoTools needs a similar number of leaf nodes (171 nodes) to prove that White wins for the position after **D1** is played. GoTools is therefore believed to try **D1** first and White is proven

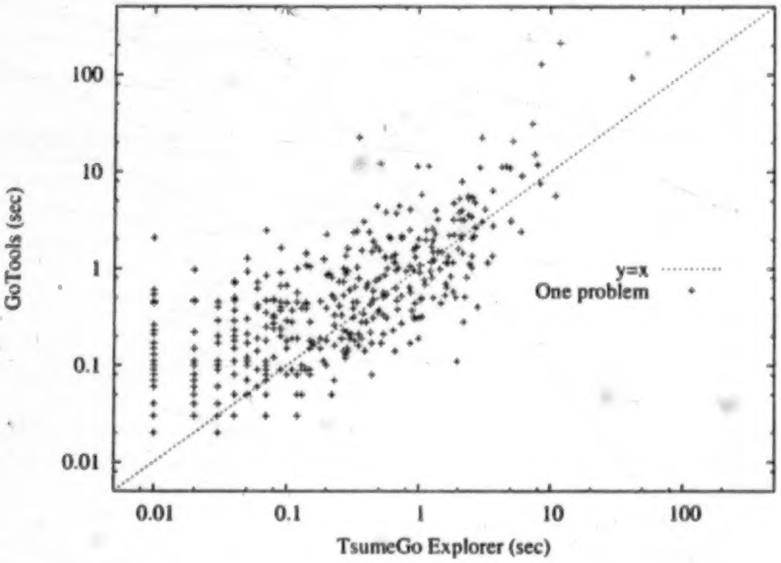


Figure 6.6: Comparison of execution time for individual instances in LV6.14.

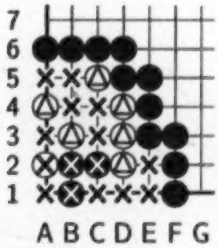


Figure 6.7: A position that GoTools solves more quickly than TsumeGo Explorer (P1031246, White lives with **D1**).

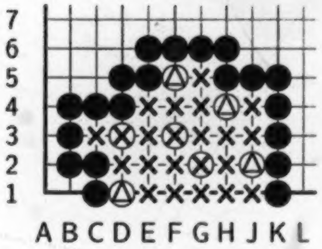


Figure 6.8: A position that TSUMEGO EXPLORER solves more quickly than GoTools (P2072215, Black kills with **E2**).

to be alive. On the other hand, TSUMEGO EXPLORER needs 0.44 seconds with 22,773 node expansions. The deepest search depth in the proof tree is 23. These numbers imply that TSUMEGO EXPLORER needs to search deeper with wider branches than GoTools for the problem. However, TSUMEGO EXPLORER usually performs better for hard problems. this is believed because of a more sophisticated search algorithm. For example, GoTools needed 211 seconds to solve the position in Figure 6.8, whereas TSUMEGO EXPLORER solves it in 11.7 seconds with 452,042 nodes.

Figures 6.9 plots the execution time for problems solved by both programs in ONEEYE. The superiority of TSUMEGO EXPLORER on hard problems is more vividly shown in this test suite. TSUMEGO EXPLORER outperforms GoTools by a large margin. Figure 6.10 shows an example. GoTools needed 121 seconds to solve this problem, whereas the problem was easy for TSUMEGO EXPLORER (0.14 seconds with 8,387 nodes). However, the Go knowledge of GoTools is sometimes very valuable. For example, Figure 6.11 with White to play is statically solved by GoTools, while TSUMEGO EXPLORER needs 3,159 nodes.

Figure 6.12 plots the problems solved only by TSUMEGO EXPLORER. Since GoTools could not solve the problems in 5 minutes, 5 minutes is plotted on the Y-axis. The difficulty for TSUMEGO EXPLORER ranges from very easy to hard. Again, the benefit of the search-based approach is shown. For example, Figure 6.13 could not be solved by GoTools within 5 minutes, while TSUMEGO

P

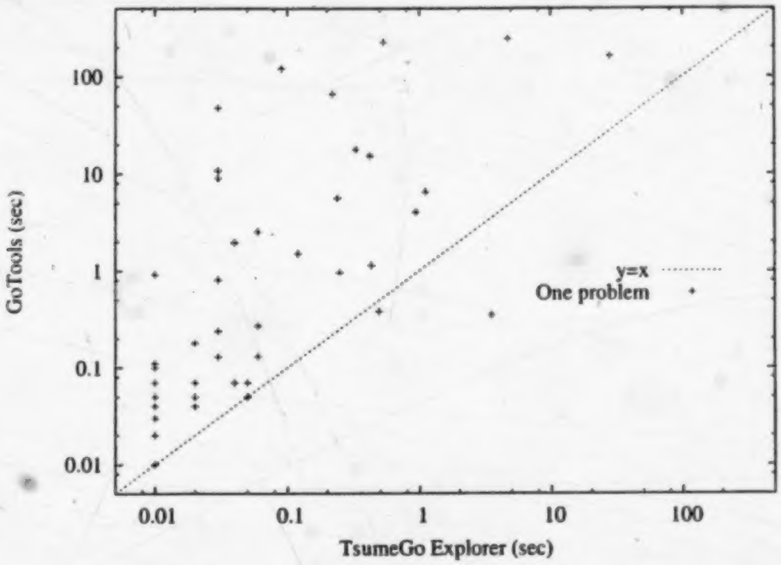


Figure 6.9: Execution time for problems solved by both programs in ONEEYE.

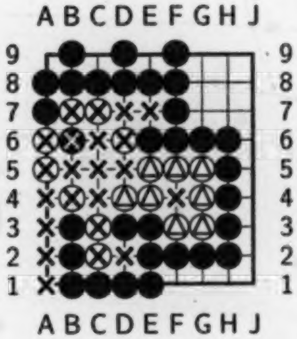


Figure 6.10: A position that is hard for GoTools but very easy for TsumeGo Explorer (oneeyec.10.sgf, Black to kill by playing at **C5**).

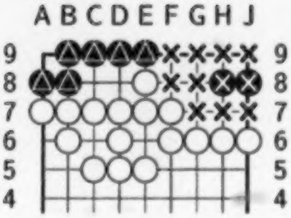


Figure 6.11: A position that GoTools solves statically (oneeyeb.10.sgf, White to kill by playing at **F9**).

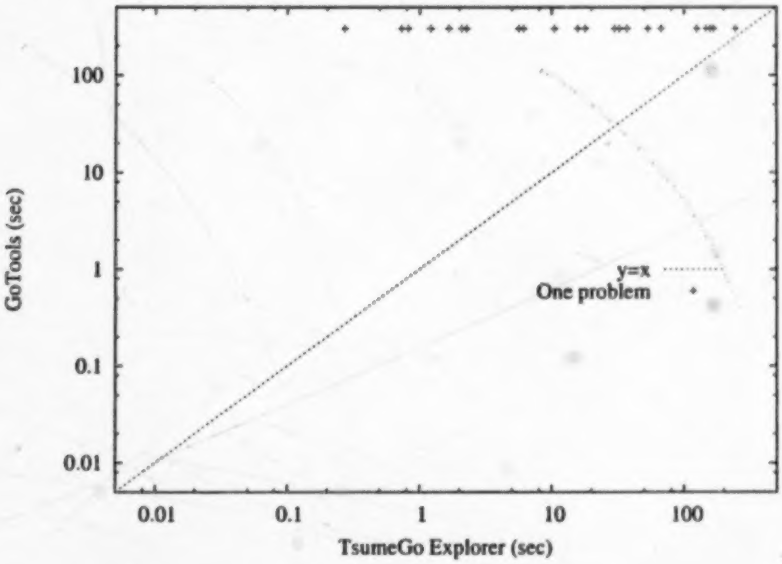


Figure 6.12: Execution time for problems solved only by TSUMEGO EXPLORER in ONEEYE.

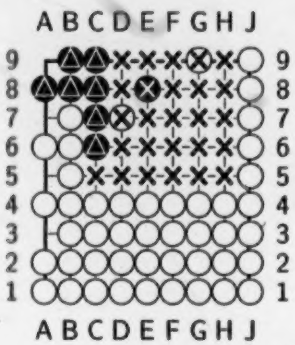


Figure 6.13: A position solved only by TsumeGo Explorer (oneeyee.9.sgf, Black to live by playing at **D8**).

Explorer solved it in 0.73 seconds with 22,616 nodes.

One advantage of df-pn is that it uses the transposition table more extensively. Only solved positions are saved in the transposition table in GoTools [90], while in df-pn proof and disproof numbers of previous iterations are stored in the transposition table to improve the order of tree expansion [56]. As a result, some problems that are very hard for GoTools are quickly solved by TsumeGo Explorer.

6.3.4 Comparison with SmartGo

Kierulf rewrote GoTools in C++ and improved when he incorporated it into SmartGo [31, 33]. This version of GoTools is about 3.4 times faster than the original on problems in LV6.14. However, it also could not solve most of the problems in ONEEYE that are unsolved by the original GoTools.

6.4 Currently Unsolved Problems

TsumeGo Explorer currently cannot solve 6 problems in the test suite. Table 6.8 presents a list of unsolved problems. There are more problems remaining unsolved than by the one-eye solver. Oneeyed.7.sgf for White to live is solved by TsumeGo Explorer with 11,874,229 nodes in 820 seconds. However, it cannot be solved within 5 minutes, because of the slower search

Table 6.8: List of unsolved problems in ONEEYE.

Problem name	Color to play
oneeyed.7.sgf	Black
oneeyed.7.sgf	White
oneeyed.9.sgf	Black
oneeyed.9.sgf	White
oneeyee.2.sgf	Black
oneeyee.11.sgf	Black

speed caused by a more complicated function to compute the heuristic distance for two eyes.

6.5 Conclusions and Future Work

The ideas invented in Chapter 5 were adapted to tsume-Go. A large reduction of the search space was achieved, thus the best tsume-Go solver was built. The current TsumeGo Explorer can solve enclosed positions with around 20 empty points in a reasonable amount of time. Judging by the size of unsolved problems, the size limit of TsumeGo Explorer seems to be between 22 and 27 empty points. This compares favorably to GoTools, which has a limit of about 14 empty points in reasonable time.

Since the knowledge added to enhance TsumeGo Explorer is quite simple, there is a lot of room for further improvement. As explained in Chapter 5, possible enhancements include recognizing larger eyes, refining connections, generalizing forced moves, and so on. For tsume-Go specific enhancements, recognizing more complicated eyes such as a half eye and one and a half eyes will be a challenging topic. This may help to decompose a tsume-Go problem in subproblems, such as making the first eye in some space, and then a second eye that can be connected to the first eye. Such a *divide and conquer approach* for the simpler case of the one-eye problem will be discussed in the next chapter.

Chapter 7

Divide and Conquer Approach to the One-Eye Problem

Divide and conquer is a way to tackle the large branching factor and deep search depth in the game of Go. This approach splits a problem into subproblems that can be independently solved, and was successful in Go endgames [52]. This chapter presents a new divide and conquer technique, called the *dynamic decomposition search (DDS)* algorithm. The results of DDS in the one-eye problem in Go show the promise of this approach. Additionally, *relaxed decomposition*, a more ambitious way of splitting positions is proposed.

7.1 Basic Idea

The basic idea of the proposed divide and conquer approach is quite simple. For example, assume that Black needs to make the second eye in Figure 7.1. A naive algorithm would generate moves at all marked points in its search. This is clearly inefficient, since the marked region is already split into two separate areas. With the exception of *ko* fights, no move played in one area can affect the result of whether there is an eye in the other area. Instead of performing a global search, a divide and conquer approach performs two local searches that can be combined into a global result. This approach can reduce the branching factor and depth of the search by a large margin.

However, if ko fights are involved in a local solution, this approach can change the ko status because formerly local ko threats become non-local. Fig-

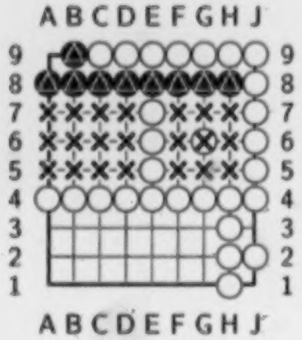
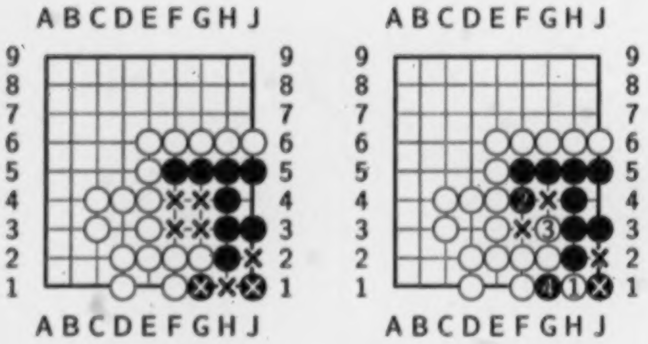


Figure 7.1: A position to which a divide and conquer approach is applicable. (Black to play.)



⑤ pass ❻ at ①

Figure 7.2: Interacting regions in ko with the divide and conquer approach.

ure 7.2 presents such a case. In this example, White can capture the ko first but Black has a local ko threat at 2. So White needs one external ko threat to win. However, if the region is divided into two parts, the solver cannot see the ko threat at 2, and the ko becomes one where Black needs an external ko threat.

To fix this problem, the solver would need to be extended to search for ko threats in all subregions whenever the result of the one eye search is a ko. This is currently not implemented. There are further complications, for example if two or more subregions end up as some kind of complex ko. For simplicity, this chapter concentrates only on eyes for which the defender does not need to fight ko. The solver correctly deals with double ko, triple ko, and so on, as long as all ko are in the same region.

Checking the number of liberties of crucial stones is important to avoid them being captured while the defender is working in one subregion. In the test example, the eye point that is given already is not marked as a region. However, in practice, it must be whether each crucial stone has more than one liberty.

7.2 The Dynamic Decomposition Search Algorithm

Let R be the region at the root position, and R_w be the *working region* that the algorithm is currently searching in. At the start, set $R_w = R$. The *dynamic decomposition search* (DDS) works as follows:

1. If there is an eye in R_w, the defender wins in region R_w.

2. Otherwise, if no eye space remains in R_w, the attacker wins in R_w.

3. Before generating moves, the region R_w is tested for a possible split into subpositions. Both safe attacker stones and crucial defender stones are used for splitting. This check is done at every newly encountered nonterminal position.

4. Suppose that a position is already partitioned into several subpositions $R_1 \cdots R_k$. If the defender is to play and a move in R_i is chosen by the search control (see the next section), then the working region R_w is restricted to R_i. Below this position, moves are generated only in R_i, or an even smaller region when further decompositions occur. This reduces the number of possible moves and the search depth to reach terminal positions. If the defender finds an eye in one of the subregions $R_1 \cdots R_n$, the defender wins. If no eye is found in any subregion, the attacker wins.

5. If the attacker is to play, all moves in R_w are tried.

Figure 7.3 presents pseudo-code of DDS, integrated with a naive AND/OR tree search algorithm. The working region can only be narrowed if the defender is to play.

7.3 Search Control

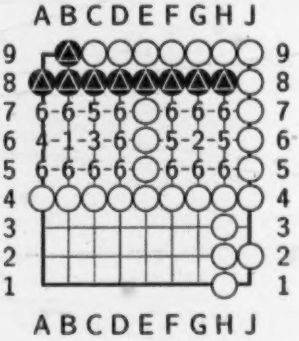
If there are several subregions, the defender must select one to expand the search in. Df-pn(r) with DDS selects the move to play based on proof and disproof numbers. This dynamically selects a most promising working region at each step. Figure 7.4 shows an example. Let the defender be a player to prove a position, and the numbers on the board be proof numbers. In this figure, Black plays at **B6**, because it has the smallest proof number. The working region is narrowed to the left side. White answers only in the left subregion after Black's **B6**. Assume that the proof number at **B6** is changed to 5 after exploring positions below Black's play at **B6**. Then, Black plays at **G6** because it becomes the smallest proof number. The working region switches to the right subregion.

7.4 Using the Transposition Table in DDS

In a normal transposition table, Zobrist hashing [92] maps a full board position to its hash key. However, DDS must distinguish between different working regions. For example, if a position contains two subregions A and B, there

```
int DDS(node $n$, region $Rw$) {
  // An eye is made in $Rw$
  if (HasOneEye($n$, $Rw$))
    return defenderWin;
  // No eye space is found in $Rw$
  if (HasNoEyeSpace($n$, $Rw$))
    return attackerWin;
  // Split $Rw$ using safe attacker stones and crucial defender stones
  RecognizeDecomposition($Rw$);
  if (IsDefender($n$)) {
    for (each child $n_{child}$ of $n$) {
      // Narrow working region
      $Ri$ = FindWorkingRegion($n_{child}$,$Rw$);
      if (DDS($n_{child}$, $Ri$)== defenderWin)
        return defenderWin;
    }
    return attackerWin;
  } else {
    for (each child $n_{child}$ of $n$) {
      if (DDS($n_{child}$, $Rw$) == attackerWin)
        return attackerWin;
    }
    return defenderWin;
  }
}
```

Figure 7.3: Pseudo-code of the DDS algorithm.



Black to play

Figure 7.4: Example of using proof numbers to select the working region.

can be three cases for move generation: only in A, only in B, and in both A and B. To differentiate these cases, DDS encodes the working region into the hash key as well.

7.5 Experimental Results

7.5.1 Setup of Experiments

Two test suites in the experiments were used. The first test suite, the *toy problem collection*, contains 13 test positions (26 problems) that are already completely or mostly split into independent problems at the root (see Appendix C.1.2). These problems were used mainly to verify that the decomposition approach works. The second test suite is the *standard problem collection* given in Appendix C. It contains 81 test positions (162 problems). Two versions of the solver, with and without dynamic decomposition search (DDS) were compared. The version without DDS, no-DDS, is the solver described in Chapter 5.

7.5.2 Results

Tables 7.1 and 7.2 compare the solving abilities of DDS and no-DDS. More positions are solved by using DDS in Toy. Moreover, all problems solved by no-DDS were also solved by DDS. On the other hand, both versions solved the same subset of problems in Standard. The improvement achieved by DDS is a factor of 66 in Toy and 1.2 in Standard in total execution time. This indicates that DDS surpasses the abilities of the previous df-pn(r) solver.

On average, DDS is about 13% slower in terms of node expansions per second (see Table 7.2). However, sometimes DDS is faster, especially in Toy (see Table 7.1). In particular, with decompositions at or near the root, DDS can concentrate on a smaller region, which speeds up basic operations such as detecting potential eye points and move generation.

Figure 7.5 compares node expansions of both solvers for each problem in the Toy test set. The number of nodes explored by DDS is plotted on the X-axis against no-DDS on the Y-axis on logarithmic scales. In points above the

Table 7.1: Performance comparison for DDS and no-DDS in Toy.

	Number of problems solved	Total time (sec) 23 problems	Total nodes expanded 23 problems	Nodes expanded per second
No-DDS	23	52	2,169,239	41,636
DDS	26	0.79	49,433	62,573
Total problems	26	-	-	-

Table 7.2: Performance comparison for DDS and no-DDS in Standard.

	Number of problems solved	Total time (sec) 157 problems	Total nodes expanded 157 problems	Nodes expanded per second
No-DDS	157	1,975	84,084,752	42,573
DDS	157	1,645	62,129,738	37,774
Total problems	162	-	-	-

diagonal DDS performed better. Except for one problem DDS expanded at most as many nodes as no-DDS, and often dramatically less. The performance of DDS scales exponentially better in the size of problems. For example, DDS solved the position in Figure 7.6 in 1,075 nodes, while no-DDS needed 334,718 nodes. This is not surprising, since all positions in this set are ideal for DDS, while no-DDS suffers from combinational explosion.

Figures 7.7 and 7.8 present the results for the standard test collection. None of these problems were designed with decomposition in mind. In contrast to Figure 7.5, there are more problems where DDS was slower. However, on average DDS explores less nodes and needs less execution time. This is especially true for the larger problems, so DDS seems to scale better. DDS sometimes improves the performance by a large margin. For example, DDS needed 360,163 nodes in 7.5 seconds for the position in Figure 7.9, whereas no-DDS explored 1,732,845 nodes in 35.6 seconds. In this position, decompositions triggered by black crucial stones and white safe stones reaching the borders of the board seem to occur frequently.

In the hard problems of this set, the percentage of nodes in which decom-

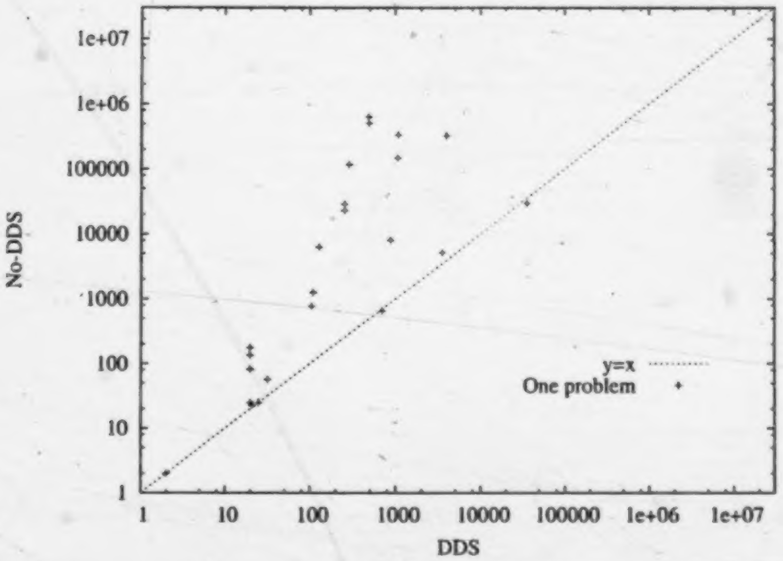


Figure 7.5: Node expansions for toy problems solved by both versions.

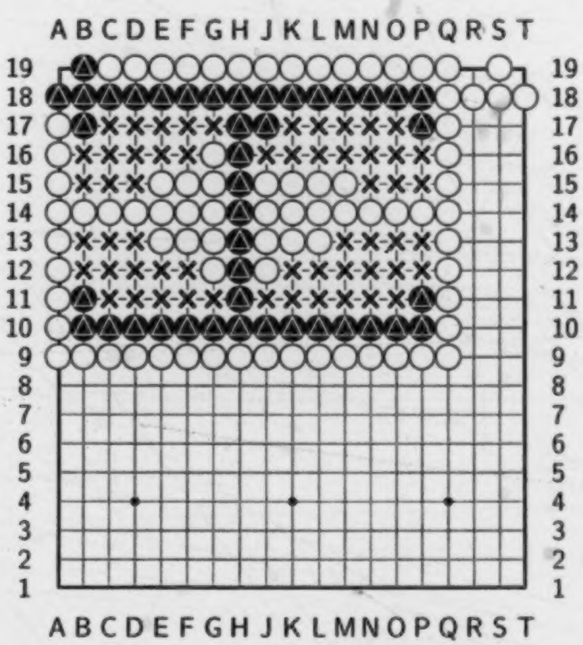


Figure 7.6: A position that DDS solved with much less nodes than no-DDS (divide-conquer.12.sgf, Black to live by playing at **O12**).

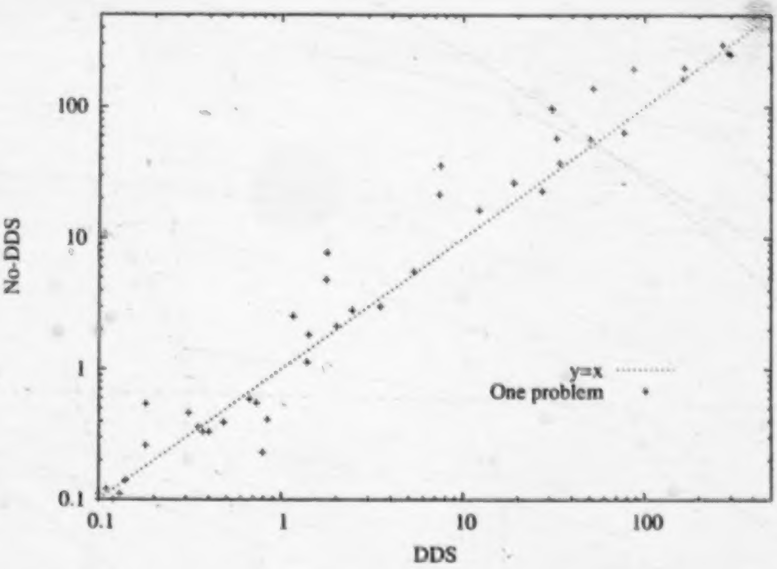


Figure 7.7: Execution time for standard problems solved by both versions.

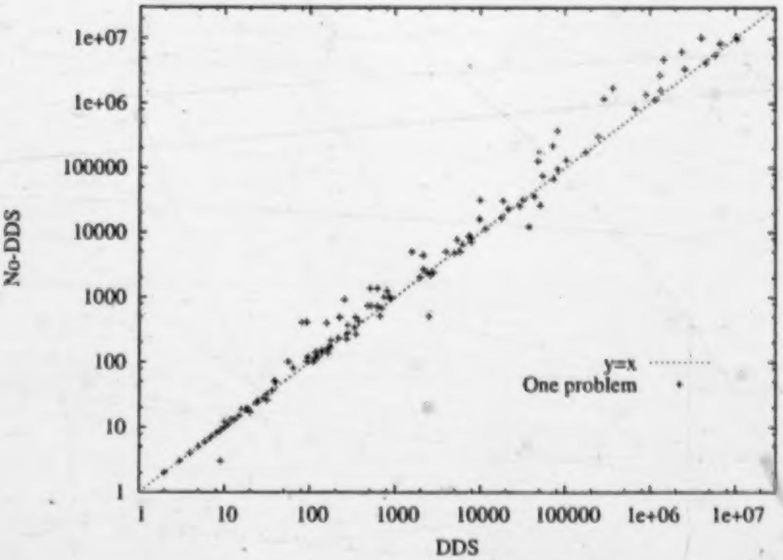


Figure 7.8: Node expansion for standard problems solved by both versions.

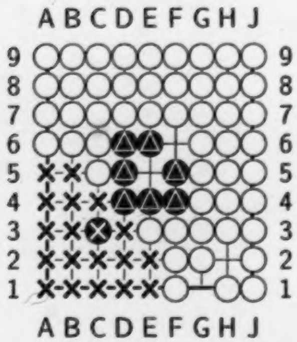


Figure 7.9: A position that DDS solved more quickly (oneeyee.1.sgf, Black lives with **B2**).

positions are possible varies from 16% to 50%. Again, for example, in Figure 7.9, in the 360,163 nodes explored by DDS, DDS detected 127,479 (35.3%) decompositions.

7.6 A Relaxed Decomposition Model

DDS is limited in the way that splits are recognized. The only points used to split positions are those occupied by safe attacker and crucial defender stones. This section introduces a less rigid decomposition that uses "almost safe" attacker stones as well. Figure 7.10 shows an example. If the two white stones marked by squares are assumed to be safe, then the area can be split into two subregions, a left subregion marked by small grey squares and a right region marked by crosses. The attacker can always make the two marked white stones safe by following a simple *miai* connection strategy: Whenever the defender plays either A or B, reply on the other point. The *relaxed decomposition* model uses such stones for splitting a position. However, during the search the case where a miai connection is attacked by the defender must be handled.

In this case, the relaxed decomposition model extends the search to the union of the affected subregions. The algorithm is explained in detail with the help of Figure 7.10. In this figure, let region R_1 consist of the empty point A and all points marked by filled squares. Region R_2 contains B and

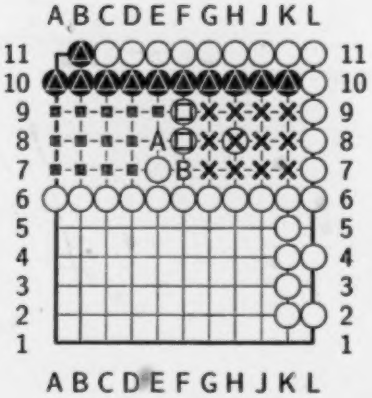


Figure 7.10: Relaxed decomposition.

all points marked by crosses. The points A and B together form a miai connection from safe attacker stones to an almost safe attacker block. Without loss of generality, assume that the defender starts playing in R_1. Then *Relaxed Decomposition Search* (RDS) defines the strategies of both players as follows:

1. If both A and B are empty, both players are restricted to play moves in R_1.
2. If either A or B is occupied by the attacker, the relaxed decomposition has changed into a normal decomposition. Both players keep playing in R_1.
3. Otherwise, if at least one of A and B contains a stone of the defender, and the other point is empty or also occupied by the defender, the region is extended and both players continue play in $R_1 \cup R_2$.

In RDS, as long as the defender does not play at A, both players stick to play in R_1. However, if the defender plays at A and the attacker does not respond at B, then the defender can invade R_2. Figure 7.11 presents pseudocode of RDS, integrated with the naive AND/OR tree search algorithm. This is similar to DDS in Figure 7.3. However, the working region can be widened, when the defender attacks a miai point but the attacker does not reply.

```
int RDS(node $n$, region $Rw$) {
  // An eye is made in $Rw$
  if (HasOneEye($n$, $Rw$))
    return defenderWin;
  // No eye space is found in $Rw$
  if (HasNoEyeSpace($n$, $Rw$))
    return attackerWin;
  // Split $Rw$ using safe attacker stones, crucial defender stones,
  // and connection to safe attacker stones
  RecognizeDecompositionUsingMiaiPoints($Rw$);
  if (IsDefender($n$)) {
    for (each child $n_{child}$ of $n$) {
      // Region can be widened
      $Ri$ = FindWorkingRegion($n_{child}$,$Rw$);
      if (RDS($n_{child}$, $Ri$)== defenderWin)
        return defenderWin;
    }
    return attackerWin;
  } else {
    for (each child $n_{child}$ of $n$) {
      if (RDS($n_{child}$, $Rw$) == attackerWin)
        return attackerWin;
    }
    return defenderWin;
  }
}
```

Figure 7.11: Pseudo-code of the RDS algorithm.

Remark: as in DDS, the defender can switch between trying moves in R_1 and R_2 at the root. This process is controlled by proof numbers.

Let us call A and B in Figure 7.10 *connection points.* the following lemma is required to prove correctness of RDS.

Lemma 7.6.1 *Assume the following:*

1. *Position P contains region R which is split into two subregions R_1 and R_2 by relaxed decomposition.*
2. *The attacker is to play in P.*
3. *Two connection points A in R_1 and B in R_2 are initially empty.*
4. *If the attacker plays at A, the defender can still create an eye unconditionally (without ko) in R_1 and the proof graph created by the defender is a DAG.*

Then, the defender can create an eye in R_1 unconditionally for the position after the attacker plays a move in R_2 for P.

Proof. Let P_1 be the node after the attacker plays a move at A for P, and P_2 be the node after the attacker plays a move in R_2 for P. The defender can be proven to make an eye for P_2 by following the winning strategy for P_1. The lemma is proven by induction on the maximum depth d of a terminal node in the proof graph of P_1.

case 0: $d = 0$ If P_1 is a terminal position, the same eye exists in both P_1 and P_2.

case 1: $d = 1$ The defender can create an eye by making move $m \neq A$ in R_1 for P_1. Since P_1 is identical to P_2 within $R_1 - \{A\}$, m is legal in P_2 and also creates an eye there.

case 2: induction step Assume that Lemma 7.6.1 holds for all $d \leq k$. The lemma is also proven to hold for $d = k + 2$. Let $m \in R_1 - \{A\}$ be the winning defender move in P_1, Q_1 be P_1's child after playing

m for P_1, and $n_1, n_2, \cdots, n_l$ be Q_1's children. Let n_1 be the node after the attacker passes for Q_1. Since $m \in R_1 - \{A\}$, m is legal for P_2. Let Q_2 be P_2's child by playing m, o_1 be Q_2's child after the attacker plays at A, $o_2 \cdots o_p$ be Q_2's children after moves in R_2, and $o_{p+1} \cdots o_{p+q}$ be Q_2's children from moves in R_1. The defender can make an eye for $o_{p+1} \cdots o_{p+q}$ by the assumption of Lemma 7.6.1. R_1 is completely separated from R_2 in o_1 and n_1. Moreover, since o_1 and n_1 are identical positions in R_1, the defender can create an eye in o_1. By induction, since o_1 has depth $d \leq k$, the defender can create an eye for $o_2 \cdots o_p$. Thus, the lemma is proven for the case of $k + 2$.

■

The following theorem guarantees the correctness of RDS in the case that an eye is found.

Theorem 7.6.1 *Assume that R is split into two subregions R_1 and R_2 by RDS. If the result of RDS shows that the defender can create an eye unconditionally (without ko) in either R_1 or R_2 and the proof graph created by RDS is a DAG, then that eye can always be made against any attacker strategy in $R_1 \cup R_2$.*

Proof. In the following, the proof graph created by RDS is called the *RDS proof graph*, and a proof graph for the whole region $R_1 \cup R_2$ an *original proof graph*.

The proof shows that each RDS proof graph can be converted into an original proof graph, for the case where the RDS proof graph is a DAG. It uses induction on the depth of a terminal node in the RDS proof graph. The first case of RDS is explained with the help of Figure 7.10. The other two cases are trivial, because searching either with completely separated subregions or with the whole region is performed in those cases.

Assume without loss of generality that the first defender move is in R_1. As above, if the RDS graph contains only a terminal node, an eye already exists and the RDS proof graph also works as an original proof graph.

Otherwise, let n be the root of the RDS proof graph. Assume that by induction n's descendants in the RDS graph have been converted to original proof graphs.

- If n is an OR node, n's move m leading to n's child n_c in the RDS proof graph is also legal for searching in $R_1 \cup R_2$. n_c's RDS proof graph can be converted to n_c's original proof graph by the induction assumption. Hence, n's original proof graph can be constructed by adding a branch m from n to n_c's original proof graph.
- If n is an AND node, assume that n's children $n_{c_1}, \cdots, n_{c_k}$ in R_1 have proof graphs. Let n_{c_1} be n's child after the attacker plays a move at A, $n_{c_{k+1}} \cdots n_{c_l}$ be n's children by playing in R_2. It is required to prove that $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs. n_{c_1} guarantees that an eye can be made in R_1, since R_1 is completely separated from R_2. By applying Lemma 7.6.1 to $n_{c_{k+1}} \cdots n_{c_l}$ based on n_{c_1}'s proof graph, the defender can make an eye for $n_{c_{k+1}} \cdots n_{c_l}$. Hence, $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs.

■

A conjecture is that Lemma 7.6.1 and Theorem 7.6.1 also hold for cyclic graphs in the case where the eye can be made unconditionally. However, a different approach is required to prove this conjecture for cyclic graphs, because a property of DAGs was used in the induction proof: it uses the fact that children have a height that is at least 1 smaller than their parents. This property does not hold for cycles.

RDS can split positions more frequently than DDS. The approach can be generalized to more than two relaxed split subregions, as long as all the miai connections to safe attacker stones are disjoint. However, the completeness of the relaxed decomposition algorithm is not known yet. To prove that no eye is possible, the worst-case scenario might require re-searches in the whole region. In this case, an efficient re-search strategy must be devised.

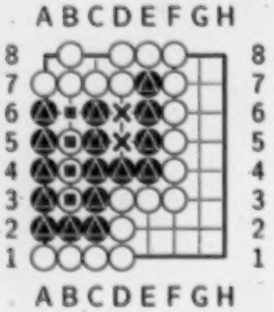


Figure 7.12: Decomposition for tsume-Go.

7.7 Conclusions and Future Work

The chapter presented a method that dynamically decomposes a position into sub-positions during search. The results of this dynamic decomposition search are encouraging. In many problems, DDS is able to reduce the search space, thereby enabling the one-eye solver to solve hard problems more quickly. However, the current version of DDS is limited to dealing with ko fights only in the same subregion. To overcome this problem, a detailed ko status and ko threat status of each divided region must be detected and combined. Additionally, investigating relaxed decomposition search is a challenging topic from both theoretical and practical points of view. Furthermore, splitting a position in a more aggressive way such as by using divider patterns [53] is an interesting extension of this research topic. Finally, applying the ideas to tsume-Go is also a challenging topic. In tsume-Go, the decomposition will be more complicated. Suppose that a region is split into two completely separated rooms A and B. There are several possibilities to be considered, such as making (1) two eyes at A, (2) one eye at A and the other eye at B, (3) two eyes at B, or (4) one eye either at A or at B if one eye already exists. In case of tsume-Go, local searches must distinguish between sente* and gote*. For example, the position in Figure 7.12 has two subregions, a left subregion R_1 marked by small grey squares and a right subregion R_2 marked by crosses. Move **B6** in R_1 makes one and a half eye [45] if Black plays first. A half eye is made in

R_2. Therefore, Black can live with **B6**, since White cannot play both **B4** and **D6**. To solve such a problem, with two separate searches in R_1 and R_2, return values of regions [45] such as "1.5 eyes" or "0.5 eyes" must be recognized by the search.

Chapter 8

Conclusions and Future Work

This chapter concludes by summarizing the research issues and the achievements. Finally, future work is discussed.

8.1 Conclusions

The focus of this thesis was to develop efficient and correct search algorithms for domains involving repetitions. The contributions of the thesis are:

- The GHI problem is a notorious problem that causes game-playing programs to falsely regard proven positions as disproven or vice versa. A solution to the GHI problem was presented (Chapter 3). Theoretical results showed that the GHI solution is correct. Experimental results in $\alpha\beta$ and df-pn in checkers and Go indicated that this approach is efficient, general and practical.

- The basic df-pn algorithm [56] has a problem with computing proof and disproof numbers that can cause it to loop infinitely in domains with repetitions. The thesis presented a modified version of df-pn, called df-pn(r), which is suitable for such domains (Chapter 4). Experiments in Go and checkers showed that df-pn(r) improves the solving abilities of the solvers.

- Based on df-pn(r), simple domain-dependent knowledge was added to the one-eye and tsume-Go solvers (Chapters 5 and 6). These enhance-

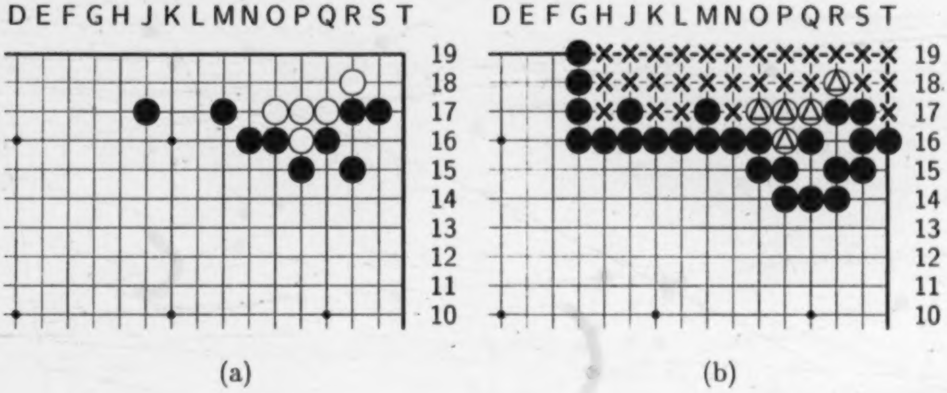

D E F G H J K L M N O P Q R S T
19
18
17
16
15
14
13
12
11
10
(a)
D E F G H J K L M N O P Q R S T
19
18
17
16
15
14
13
12
11
10
(b)

Figure 8.1: An example in Igo Hatsuyo-ron (White to live) and its enclosed version.

ments improved the solvers by an order of magnitude. Thus, this approach succeeded in developing the first powerful one-eye solver as well as TsumeGo Explorer, the currently best tsume-Go solver.

- The decomposition search algorithm that has been successful in Go endgames was extended and applied to the one-eye problem (Chapter 7). This approach reduced the search space on average, and improved the performance of the one-eye solver. Relax decomposition search, a more relaxed way to split a position into sub-positions was investigated and a theorem on correctness was proven.

8.2 Future Work

There are still many unexplored topics. Many are already mentioned in the individual chapters:

- Now that correctness of a modified df-pn algorithm is guaranteed even in the presence of repetitions, it is time to investigate completeness of df-pn. An open question is whether df-pn(r) in Chapters 3 and 4 can in principle solve any problem involving repetitions, if an unlimited amount of time is given.

- Invent more effective methods to solve harder problems. Currently I am trying to build a tsume-Go solver that can solve some modified problems from Igo Hatsuyo-ron [28], enclosed by Martin Müller (see an example of an original and an enclosed problem in Figure 8.1). Igo Hatsuyo-ron is one of the hardest test collections in the literature, and is beyond the capability of current tsume-Go solvers. I believe that some of these problems are solvable with smarter knowledge and sophisticated algorithms to split positions. Future research also includes recognition of larger eyes, and generalization of forced moves.

- One more practical extension is to write a solver for open positions, in which stones do not have to be surrounded by opponent's safe stones (see Figure 8.1(a) again). Open tsume-Go problems can become dramatically harder than enclosed problems. Wolf describes some reasons in [89]. The essential reason is the difference in the number of moves to be considered between closed and open positions. For a closed position, moves can be restricted to inside a region. On the other hand, all legal moves potentially have to be investigated on the whole board for an open position, since a slight difference might influence the whole board. One interesting open question is how accurately and efficiently tsume-Go solvers can solve problems with open boundaries. Moves are probably required to be heuristically restricted, or threat-based approaches such as [15, 82] are used to find a set of moves that guarantees correctness.

Bibliography

[1] S. G. Akl and M. M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Annual Conference Proceedings*, pages 466–473, 1977.

[2] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Department of Computer Science, University of Limburg, 1994.

[3] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12:7–23, 1996.

[4] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[5] E. B. Baum and W. D. Smith. Best play for imperfect players and game tree search; part I - theory. Technical report, NEC Research Institute, 1995. Available at `http://citeseer.nj.nec.com/baum95best.html`.

[6] D. B. Benson. Life in the game of Go. *Information Sciences*, 10:17–29, 1976.

[7] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways.* Academic Press, 1982.

[8] H. Berliner and C. Ebeling. Pattern knowledge and search: The SUPREME architecture. *Artificial Intelligence*, 38(2):161–198, 1989.

[9] B. Bouzy and T. Cazenave. Computer Go: An AI-oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[10] D. M. Breuker, L. V. Allis, and H. J. van den Herik. How to mate: Applying proof-number search. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 251–272, 1994.

[11] D. M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *International Computer Chess Association Journal*, 19(3):175–180, 1996.

[12] D. M. Breuker, H. J. van den Herik, J. W. H. M. Uiterwijk, and L. V. Allis. A solution to the GHI problem for best-first search. *Theoretical Computer Science*, 252(1-2):121–149, 2001.

[13] M. Campbell. The graph-history interaction: On ignoring position history. In *1985 Association for Computing Machinery Annual Conference*, pages 278–280, 1985.

[14] M. Campbell, A. Joseph Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[15] T. Cazenave. Abstract proof search. In T. A. Marsland and I. Frank, editors, *Computers and Games (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2001.

[16] T. Cazenave. Generation of patterns with external conditions for the game of Go. In *Advances in Computer Games 9*, pages 275–293, 2001.

[17] T. Cazenave. A generalized threats search algorithm. In *Computers and Games (CG2002)*, volume 2883 of *Lecture Notes in Computer Science*, pages 75–87. Springer, 2003.

[18] K. Chen and Z. Chen. Static analysis of life and death in the game of Go. *Information Sciences*, 121:113–134, 1999.

[19] J. Condon and K. Thompson. Belle chess hardware. In M. Clarke, editor, *Advances in Computer Chess 3*, pages 45–54. Pergamon Press, 1982.

[20] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[21] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[22] D. Dyer. An eye shape library for computer Go. Information is available at `http://www.andromeda.com/people/ddyer/go/shape-library.html`.

[23] S. L. Epstein. Game playing: The next moves. In *16th National Conference on Artificial Intelligence (AAAI'99)*, pages 987–993. AAAI Press, 1999.

[24] R. Feldmann. *Spielbaumsuche auf Massiv Parallelen Systemen*. PhD thesis, University of Paderborn, 1993. English translation titled *Game Tree Search on Massively Parallel Systems* is available at `http://citeseer.nj.nec.com/feldmann93game.html`.

[25] D. Fotland. Static eye analysis in "The Many Faces of Go". *ICGA Journal*, 25(4):203–210, 2002.

[26] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt chess program. In *Proceedings of ACM Fall Joint Computing Conference*, volume 31, pages 801–810, San Francisco, 1967. Springer-Verlag.

[27] F.-h. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie Mellon University, 1990. Also available as a technical report CMU-CS-90-108.

[28] E. Inoue. *Igo Hatsuyo-ron*. Yutopian Enterprises, 1997. ISBN: 1889554022.

[29] T. Ito, Y. Kawano, M. Seo, and K. Noshita. Recent progress in solving tsume-shogi by computers. *Journal of the Japanese Society of Artificial Intelligence*, 10(6):853–859, 1995. In Japanese.

[30] Y. Kawano. Using similar positions to search game trees. In Richard J. Nowakowski, editor, *Games of No Chance*, volume 29 of *MSRI Publications*, pages 193–202. Cambridge University Press, 1996.

[31] A. Kierulf. The SmartGo program. `http://www.smartgo.com/`.

[32] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* PhD thesis, Swiss Federal Institute of Technology Zürich, 1990.

[33] A. Kierulf. Personal communication, 2005.

[34] A. Kishimoto. About the tsume-shogi solver of ISshogi (in Japanese). In *Computer Shogi Association Bulletin*, volume 16, pages 41–46. Computer Shogi Association, 2003. Available at `http://www.cs.ualberta.ca/~kishi/publication.html`.

[35] A. Kishimoto. A correct algorithm to prove no-mate positions in shogi. In *9th Game Programming Workshop in Japan (GPW-04)*, pages 1–8, 2004. In Japanese.

[36] A. Kishimoto. Search techniques for the one-eye and tsume-Go problems. In *9th Game Programming Workshop in Japan (GPW-04)*, page 71, 2004. Invited talk. In Japanese.

[37] A. Kishimoto and M. Müller. Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games. Many Games, Many Challenges*, pages 125–141. Kluwer Academic Publishers, 2003.

[38] A. Kishimoto and M. Müller. A solution to the GHI problem for depth-first proof-number search. In *7th Joint Conference on Information Sciences*, pages 489–492, 2003.

[39] A. Kishimoto and M. Müller. A general solution to the graph history interaction problem. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 644–649. AAAI Press, 2004.

[40] A. Kishimoto and M. Müller. A solution to the GHI problem for depth-first proof-number search. *Information Sciences*, 2005. To appear.

[41] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[42] R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[43] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[44] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess*, volume 7, pages 135–162, 1994.

[45] H. Landman. Eyespace values in Go. In R. Nowakowski, editor, *Games of No Chance*, pages 227–257. Cambridge University Press, 1996.

[46] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *The 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, 1997.

[47] T. A. Marsland. Relative performance of alpha-beta implementations. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'83)*, pages 763–766, Karlsruhe, Germany, 1983.

[48] A. Martelli and U. Montanari. Additive AND/OR graphs. In *3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 1–11, 1973.

[49] D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[50] J. McCarthy. Chess as the Drosophila of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 227–237. Springer-Verlag, 1990.

[51] M. Müller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In Hitoshi Matsubara, editor, *Game Programming Workshop in Japan '97*, pages 80–86, Tokyo, Japan, 1997. Computer Shogi Association.

[52] M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'99)*, volume 1, pages 578–583, 1999.

[53] M. Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

[54] A. Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Complex Games Lab Workshop*, pages 40–45, 1998.

[55] A. Nagai. A new depth-first search algorithm for AND/OR trees. Master's thesis, Department of Information Science, University of Tokyo, 1999.

[56] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications.* PhD thesis, Department of Information Science, University of Tokyo, 2002.

[57] A. Nagai and H. Imai. Application of df-pn$^+$ to Othello endgames. In *Game Programming Workshop in Japan '99*, pages 16–23, Kanagawa, Japan, 1999.

[58] A. Nagai and H. Imai. Proof for the equivalence between some best-first algorithms and depth-first algorithms for AND/OR trees. In *KOREA-JAPAN Joint Workshop on Algorithms and Computation*, pages 163–170, 1999.

[59] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co, Palo Alto, CA, 1980.

[60] A. J. Palay. *Searching with Probabilities.* PhD thesis, Carnegie Mellon University, 1983.

[61] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, 87(1-2):255–293, 1996.

[62] R. Popma and L. V. Allis. Life and death refined. In J. van den Herik and L. V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3*, pages 157–164. Ellis Horwood, 1992.

[63] M. Pratola and T. Wolf. Optimizing GoTools' search heuristics using genetic algorithms. *International Computer Games Association Journal*, 26(1):28–49, 2003.

[64] A. Reinefeld. An improvement of the Scout algorithm. *International Computer Chess Association Journal*, 6(4):4–14, 1983.

[65] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (Second Edition).* Prentice Hall, 2002.

[66] M. Sakuta. *Deterministic Solving of Problems with Uncertainty.* PhD thesis, Department of Science and Engineering, Shizuoka University, 2001.

[67] A. L. Samuel. Some studies in machine learning. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[68] J. Schaeffer. Chinook home page. `http://www.cs.ualberta.ca/~chinook/`.

[69] J. Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[70] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

[71] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag, 1997.

[72] J. Schaeffer. A gamut of games. *AI Magazine*, 22(3):29–46, 2001.

[73] J. Schaeffer, Y. Björnsson, N. Burch, P. Lu, and S. Sutphen. Building the Checkers 10-piece endgame databases. In *Advances in Computer Games. Many Games, Many Challenges*, pages 193–210. Kluwer Academic Publishers, 2003.

[74] J. Schaeffer and A. Plaat. Kasparov versus Deep Blue: The re-match. *International Computer Chess Association Journal*, 20(2):95–101, 1997.

[75] M. Schijf, L. V. Allis, and J. W. H. M. Uiterwijk. Proof-number search and transpositions. *International Computer Chess Association Journal*, 17(2):63–74, 1994.

[76] Sensei's library: Go terms. `http://senseis.xmp.net/?GoTerms`.

[77] M. Seo. The C* algorithm for AND/OR tree search and its application to a tsume-shogi program. Master's thesis, Department of Information Science, University of Tokyo, 1995.

[78] D. J. Slate and L. R. Atkin. CHESS 4.5 - the Northwestern University chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[79] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[80] Y. Tanase. Algorithms in ISshogi. In Hitoshi Matsubara, editor, *Advances in Computer Shogi 3*, pages 1–14. Kyouritsu Shuppan Press, 2000. In Japanese.

[81] K. Thompson. 6-piece endgames. *International Computer Chess Association Journal*, 19(4):215–226, 1996.

[82] T. Thomsen. Lambda-search in game trees – with application to Go. In T. A. Marsland and I. Frank, editors, *Computers and Games (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2001.

[83] E. C. D. van der Werf, H. J. van den Herik, and J. W. H. M. Uiterwijk. Solving Go on small boards. *International Computer Games Association Journal*, 26(2):92–107, 2003.

[84] R. Vilà and T. Cazenave. When one eye is sufficient: A static approach classification. In *Advances in Computer Games. Many Games, Many Challenges*, pages 109–124. Kluwer Academic Publishers, 2003.

[85] T. Wolf. GoTools: 40,000 problems database. `http://www.qmw.ac.uk/~ugah006/gotools/t.wolf.gotools.problems.html`.

[86] T. Wolf. Investigating tsumego problems with RisiKo. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2*, pages 153–160. Ellis Horwood, 1991.

[87] T. Wolf. Quality improvements in tsume go mass production. In *Second Go and Computer Science Workshop*, 1993.

[88] T. Wolf. The program GoTools and its computer-generated tsume Go database. In Hitoshi Matsubara, editor, *Game Programming Workshop in Japan '94*, pages 84–96, Tokyo, Japan, 1994. Computer Shogi Association.

[89] T. Wolf. About problems in generalizing a tsumego program to open positions. In H. Matsubara, editor, *Game Programming Workshop in Japan '96*, pages 20–26, 1996.

[90] T. Wolf. Forward pruning and other heuristic search techniques in tsume Go. *Information Sciences*, 122(1):59–76, 2000.

[91] H. Yamashita. Aya. Available at `http://plaza15.mbn.or.jp/~yss/`.

[92] A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.

Appendix A

Proof that Df-pn Loops Forever in the Example of Figure 4.2

Figure A.1 shows the crucial steps in running df-pn on the example from Figure 4.2 on page 75. Let k be the number of visits to node C in Figure 4.2, l be a positive integer, and $\mathbf{pn}_k(n)$, $\mathbf{dn}_k(n)$ be proof and disproof numbers of node n at the kth visit to C. Let $X > Y$ mean that X is preferred to Y. Assume the following ordering of nodes for tie breaking to choose a child with tied (dis)proof numbers:

- $B > C$,
- $F > G$,
- $I > H$,
- $J > K$,
- $N > M$, and
- $I > P$.

To prove that df-pn never expands P in Figure A.1(a), first the fact that the following equations hold is shown:

$$\begin{aligned}\mathbf{pn}_k(X) &= 1 \quad \text{for all } k \text{ and all } X \in \{F, G, H, I, J, K, L, M, N, O, P\}\\ \mathbf{dn}_k(X) &= 1 \quad \text{for all } k \text{ and all } X \in \{H, K, M, P\}\end{aligned}$$

$$\mathbf{dn}_k(F) = \begin{cases} 1, & (\text{for } k = 1) \\ k, & (\text{for } k = 2l) \\ k-1, & (\text{for } k = 2l+1) \end{cases}$$

$$\mathbf{dn}_k(G) = \begin{cases} k, & (\text{for } k = 2l-1) \\ k-1, & (\text{for } k = 2l) \end{cases}$$

$$\mathbf{dn}_k(I) = \mathbf{dn}_k(L) = \begin{cases} 1, & (\text{for } k = 1) \\ k-1, & (\text{for } k = 2l) \\ k-2, & (\text{for } k = 2l+1) \end{cases}$$

$$\mathbf{dn}_k(J) = \mathbf{dn}_k(O) = \begin{cases} 1, & (\text{for } k = 1, 2) \\ k-1, & (\text{for } k = 2l+1) \\ k-2, & (\text{for } k = 2l+2) \end{cases}$$

$$\mathbf{dn}_k(N) = \begin{cases} 1, & (\text{for } k = 1, 2, 3) \\ k-2, & (\text{for } k = 2l+2) \\ k-3, & (\text{for } k = 2l+3). \end{cases}$$

Let $\mathbf{th}_{\mathbf{pn}}(n)$ and $\mathbf{th}_{\mathbf{dn}}(n)$ be the threshold of the proof and disproof numbers at n. Now the above equations is proven by induction on k.

- (Case $k \leq 5$) Figure A.1(b)-(i) gives a trace of df-pn in Figure A.1(a), which illustrates that P is not explored in case of $k \leq 4$. In this figure, (c), (e), (g), and (i) correspond to $k = 1$, $k = 2$, $k = 3$, and $k = 4$. Clearly the proof and disproof numbers for each node in this figure hold. An analogous proof can be given for the case of $k = 5$.
- Assume that the equations are satisfied at the kth visit. They are proven for $k + 1$ by tracing the search graph.
  - (Case $k = 2l + 2$) At the kth visit to C, $\mathbf{pn}_k(F) = \mathbf{pn}_k(G) = 1$ and $\mathbf{dn}_k(F) = k > \mathbf{dn}_k(G) = k - 1$. G is, therefore, chosen with $\mathbf{th}_{\mathbf{pn}}(G) = 1$ and $\mathbf{th}_{\mathbf{dn}}(G) = k + 1$. At G, $\mathbf{th}_{\mathbf{pn}}(G) = 2 \geq \min(\mathbf{pn}_k(J), \mathbf{pn}_k(K)) = 1$ and $\mathbf{th}_{\mathbf{dn}}(G) = k + 1 \geq \mathbf{dn}_k(J) + \mathbf{dn}_k(K) = k - 2 + 1 = k - 1$, G is expanded. Since $\mathbf{pn}_k(J) = \mathbf{pn}_k(K) = 1$, J is chosen with $\mathbf{th}_{\mathbf{pn}}(J) = 2$ and $\mathbf{th}_{\mathbf{dn}}(J) = k$. At J, O is explored since $\mathbf{th}_{\mathbf{pn}}(J) = 2 \geq \min(\mathbf{pn}_k(O)) = 1$ and $\mathbf{th}_{\mathbf{dn}}(J) = k \geq \mathbf{dn}_k(O) = k - 2$. Therefore, O is selected to expand with $\mathbf{th}_{\mathbf{pn}}(O) = 2$ and $\mathbf{th}_{\mathbf{dn}}(O) = k$. However, the termination condition is satisfied at O, because $\mathbf{dn}_k(I) + \mathbf{dn}_k(P) = k - 1 + 1 =$

$k \geq \mathbf{th}_{\mathbf{dn}}(O) = k$. Thus, P is not explored and proof and disproof numbers are backed up to C as follows:

$$\begin{aligned}
\mathbf{pn}_{k+1}(O) &= \min(\mathbf{pn}_k(I), \mathbf{pn}_k(P)) = 1 \\
\mathbf{dn}_{k+1}(O) &= \mathbf{dn}_k(I) + \mathbf{dn}_k(P) = k \\
\mathbf{pn}_{k+1}(J) &= \mathbf{pn}_{k+1}(O) = 1 \\
\mathbf{dn}_{k+1}(J) &= \mathbf{dn}_{k+1}(O) = k \\
\mathbf{pn}_{k+1}(G) &= \min(\mathbf{pn}_{k+1}(J), \mathbf{pn}_k(K)) = 1 \\
\mathbf{dn}_{k+1}(G) &= \mathbf{dn}_{k+1}(J) + \mathbf{dn}_k(K) = k + 1.
\end{aligned}$$

For the remaining nodes F, H, I, K, L, M, and N, proof and disproof numbers remain the same. Thus, the equations shown above are proven for the case of $k+1$.

- (Case $k = 2l+3$) This case is proven with an analogous discussion. At the kth visit to C, $\mathbf{pn}_k(F) = \mathbf{pn}_k(G) = 1$ and $\mathbf{dn}_k(F) = k-1 < \mathbf{dn}_k(G) = k$. F is chosen with $\mathbf{th}_{\mathbf{pn}}(F) = 2$ and $\mathbf{th}_{\mathbf{pn}}(F) = k+1$. $C \rightarrow F \rightarrow I \rightarrow L \rightarrow N$ is explored. The termination condition at N holds, since $\mathbf{dn}_k(O) = k-1 \geq \mathbf{th}_{\mathbf{dn}}(N) = k-1$. Then, proof and disproof numbers are backed up to C and the proof and disproof numbers change as in the equations above.

P is never explored and df-pn loops forever. ■

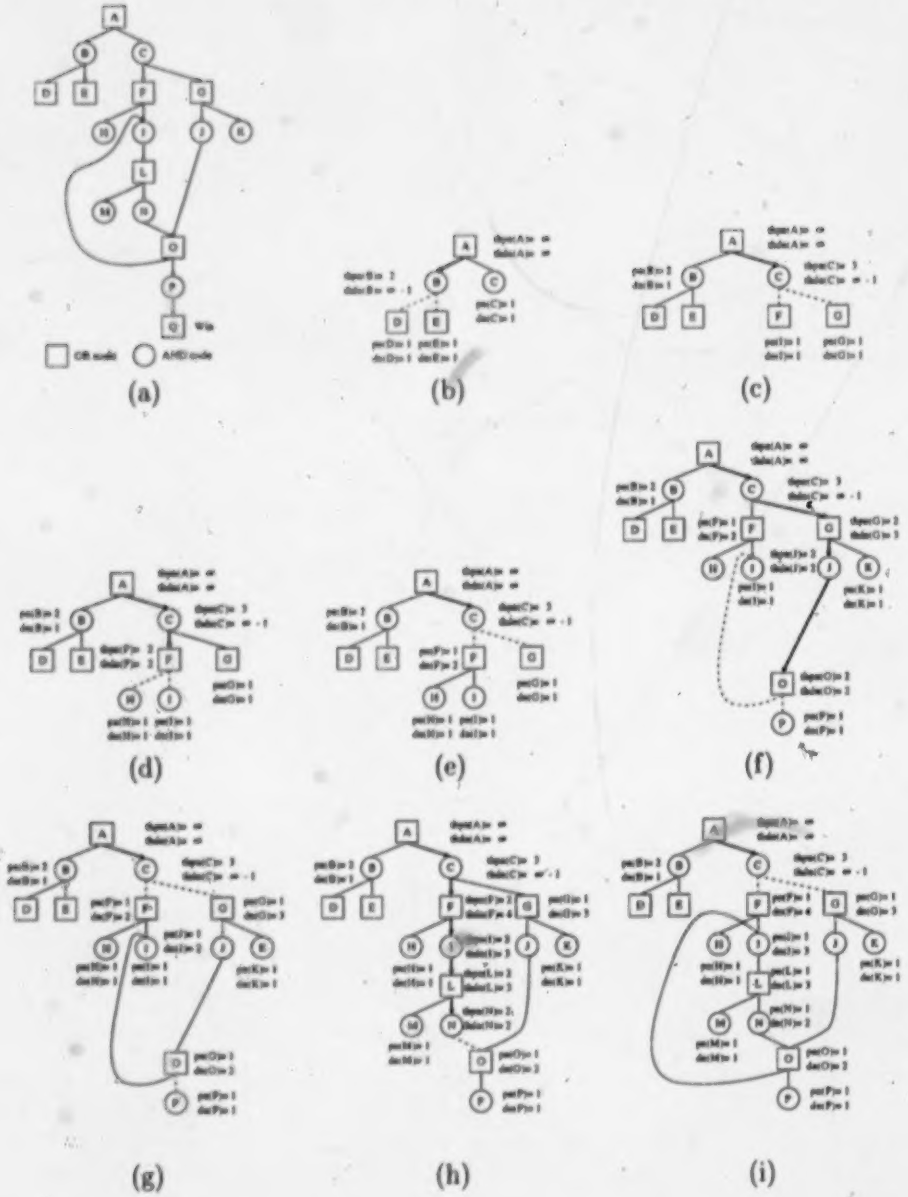


Figure A.1: An example in which df-pn loops forever.

Appendix B

Glossary

The terms used in the thesis are adapted from [76] and [53].

Block	Connected stones of the same color. See Figure B.1.
Eye	A single empty point surrounded by stones of the same color. See empty points marked by squares in Figure B.2. Stones that have two eyes are guaranteed to be alive.
Gote	Opposite of sente. See sente.
Ko	The situation in Figure B.3 is called ko, which can lead to repeated positions. In a ko situation, if one player takes the ko, the opponent may not re-capture it immediately. See the empty point marked by a square in the figure after Black plays a move.
Liberty	An empty point adjacent to a block of stones. See empty points marked by squares in B.4.
Miai	A player has two different options at his or her disposal.

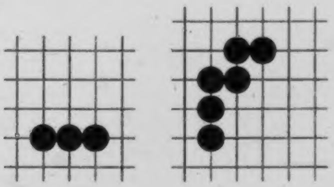


Figure B.1: Blocks.

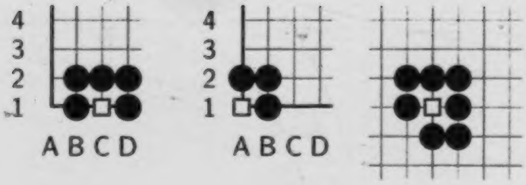


Figure B.2: Eye.

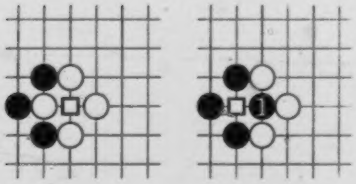


Figure B.3: Ko.

Sente In Japanese, sente means initiative. A player has sente if he or she does not have to answer the opponent's last move.

Seki Co-existence. Both black and white blocks are alive without two eyes. In Figure B.5, adapted from [76], both marked black and white stones do not have two eyes, but are alive. If either player plays *A*, the other will capture the block of that player.

Territory An area surrounded and controlled by one player. Dead opponent stones may be contained in a territory.

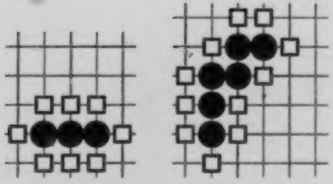


Figure B.4: Liberties.

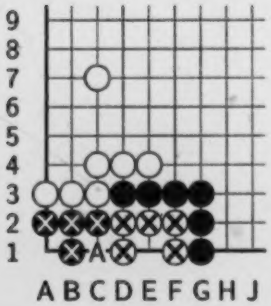


Figure B.5: Seki.

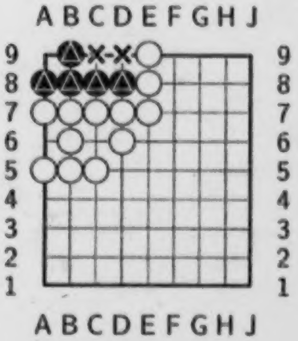
Appendix C

Positions Used for Experiments

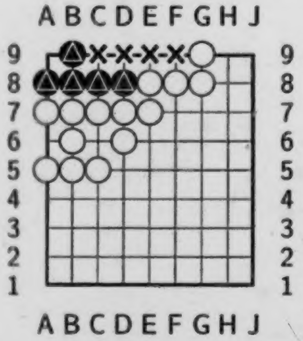
C.1 Go

C.1.1 Positions Used for All Experiments

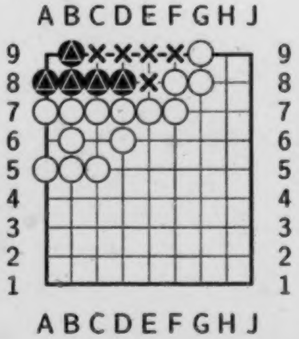
The positions are available at `http://www.cs.ualberta.ca/~games/go/oneeye`.



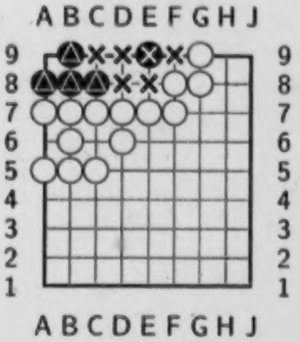
oneeye.1.sgf



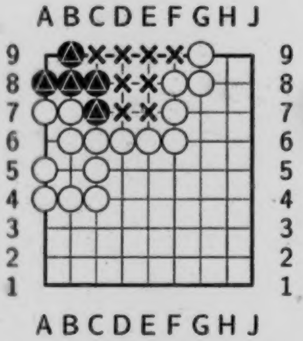
oneeye.2.sgf



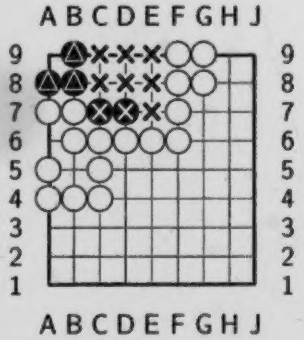
oneeye.3.sgf



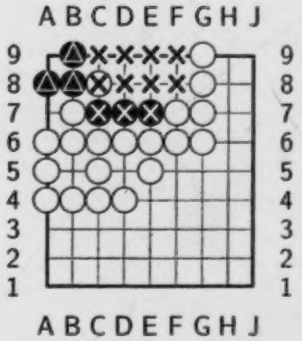
oneeye.4.sgf



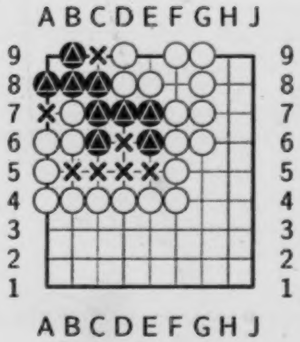
oneeye.5.sgf



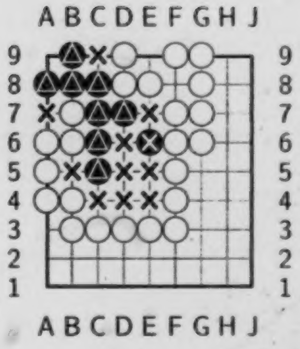
oneeye.6.sgf



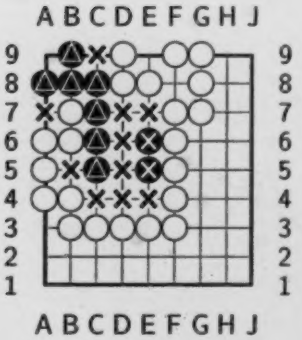
oneeye.7.sgf



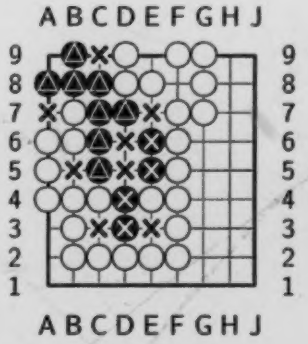
oneeye.8.sgf



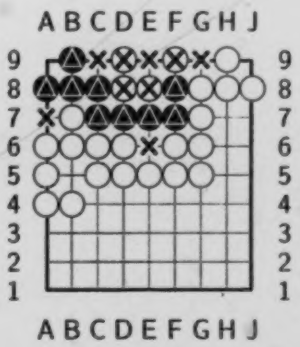
oneeye.9.sgf



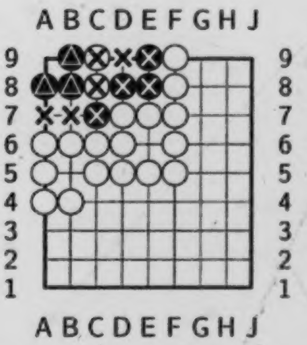
oneeye.10.sgf



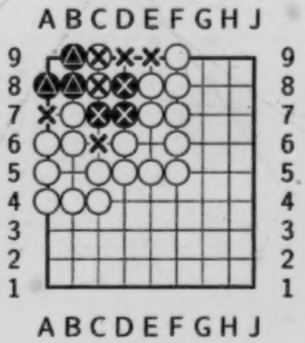
oneeye.11.sgf



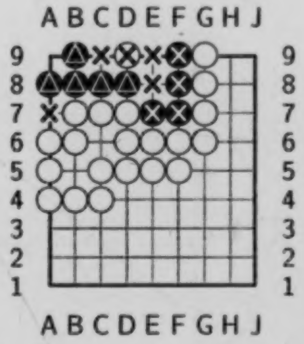
oneeye.12.sgf



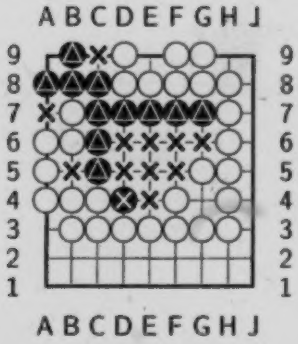
oneeye.13.sgf



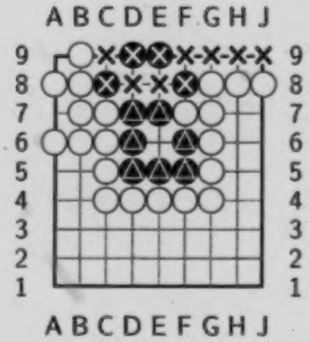
oneeye.14.sgf



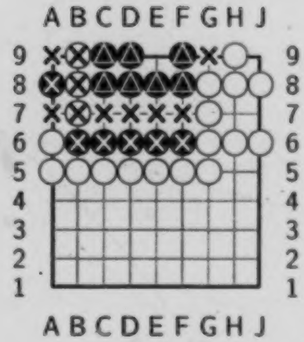
oneeye.15.sgf



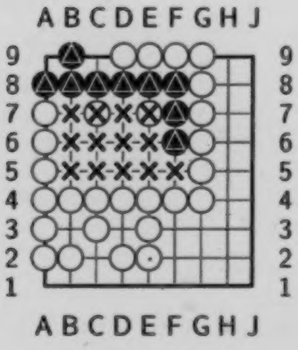
oneeye.16.sgf



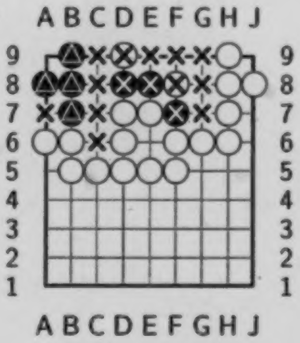
oneeye.17.sgf



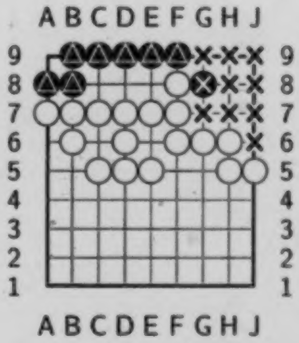
oneeye.18.sgf



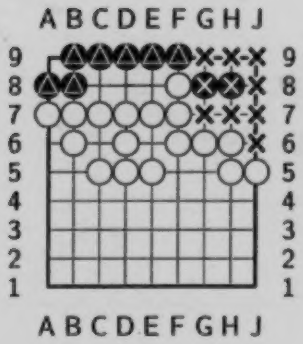
oneeye.19.sgf



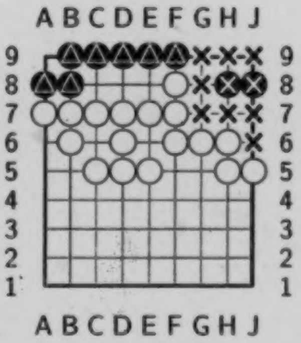
oneeye.20.sgf



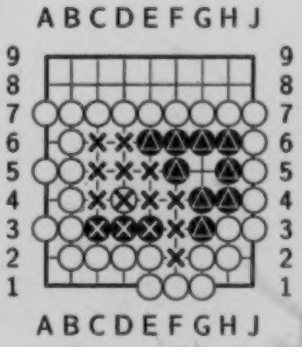
oneeyeb.1.sgf



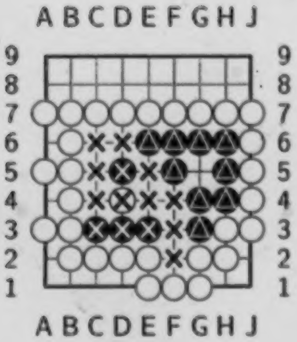
oneeyeb.2.sgf



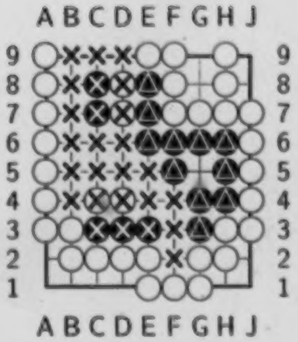
oneeyeb.3.sgf



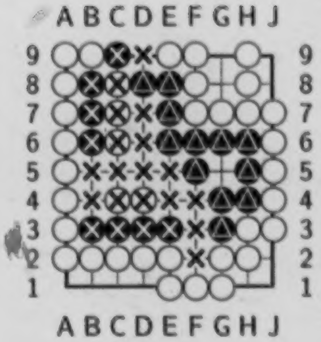
oneeyeb.4.sgf



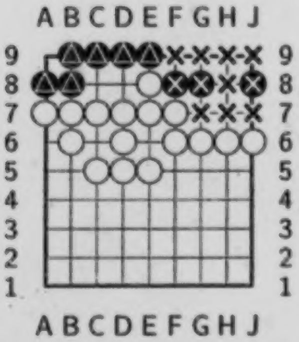
oneeyeb.5.sgf



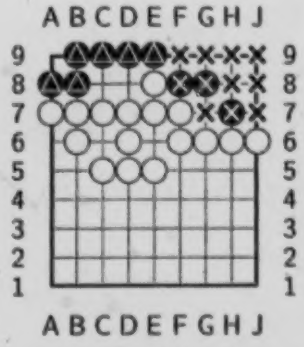
oneeyeb.6.sgf



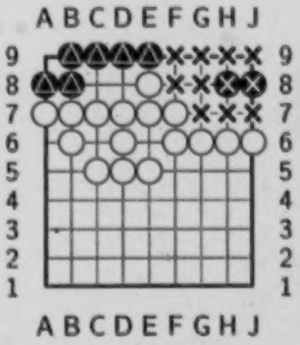
oneeyeb.7.sgf



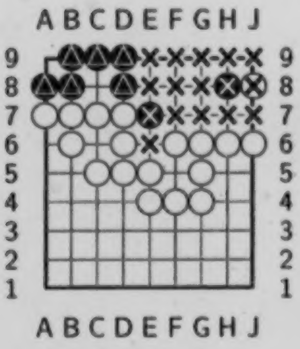
oneeyeb.8.sgf



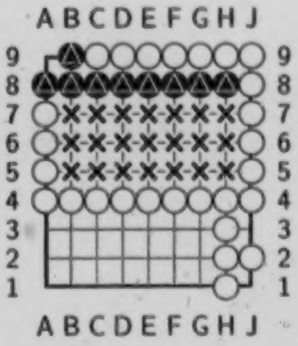
oneeyeb.9.sgf



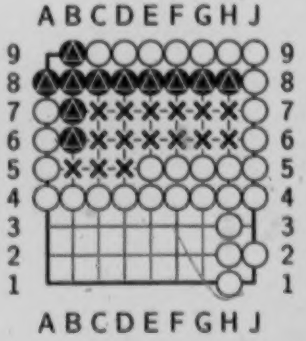
oneeyeb.10.sgf



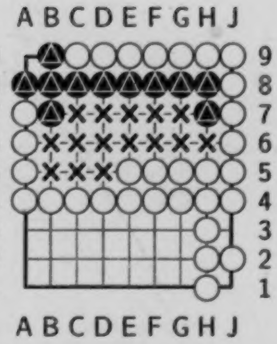
oneeyeb.11.sgf



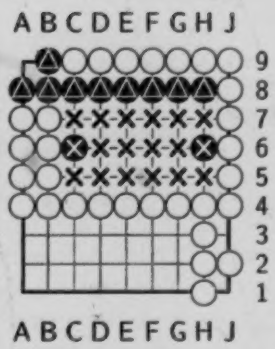
oneeyec.1.sgf



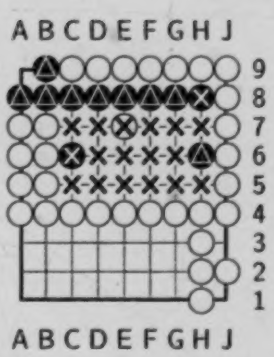
oneeyec.2.sgf



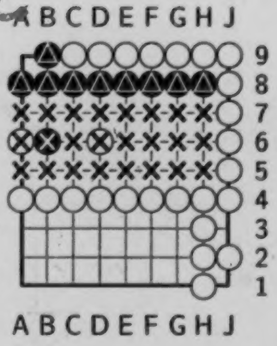
oneeyec.3.sgf



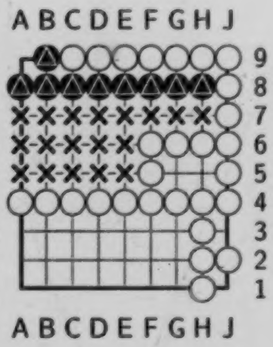
oneeyec.4.sgf



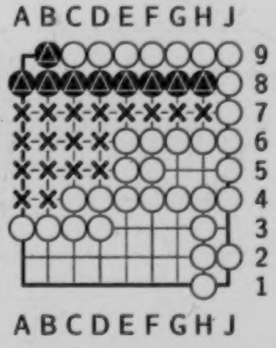
oneeyec.5.sgf



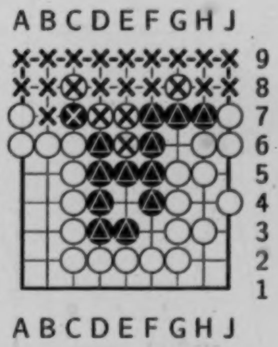
oneeyec.6.sgf



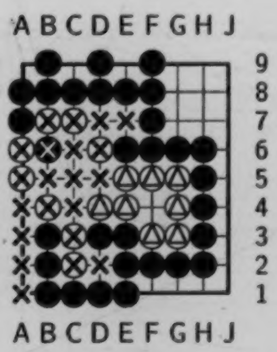
oneeyec.7.sgf



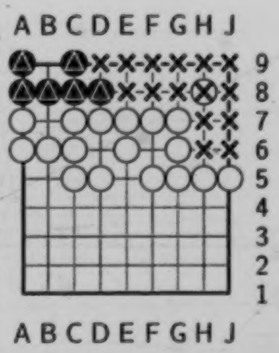
oneeyec.8.sgf



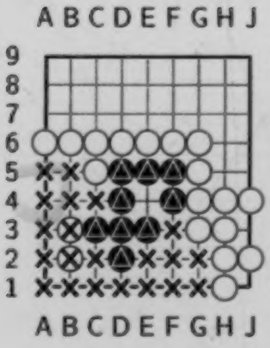
oneeyec.9.sgf



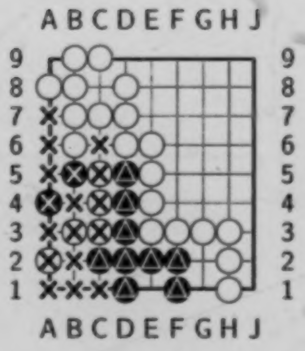
oneeyec.10.sgf



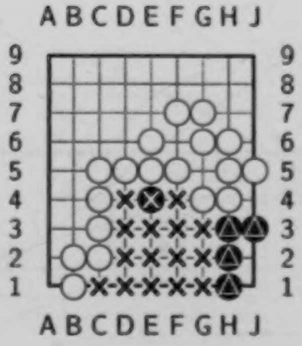
oneeyed.1.sgf



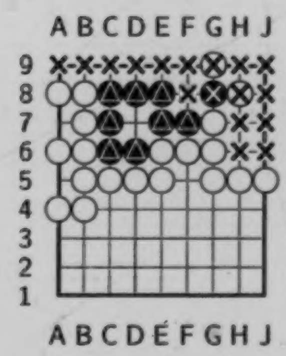
oneeyed.2.sgf



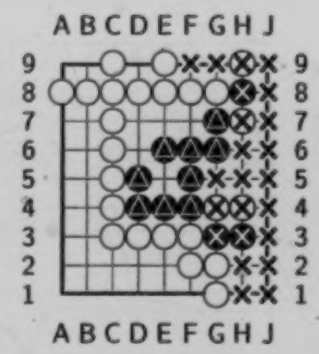
oneeyed.3.sgf



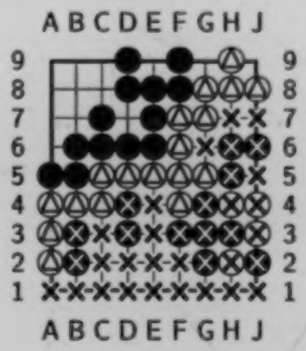
oneeyed.4.sgf



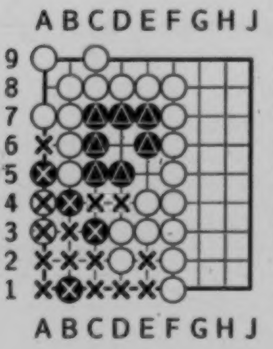
oneeyed.5.sgf



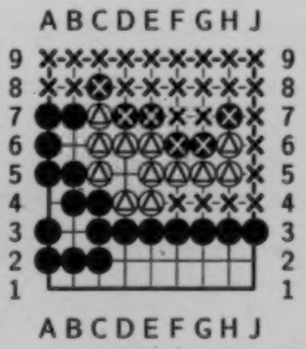
oneeyed.6.sgf



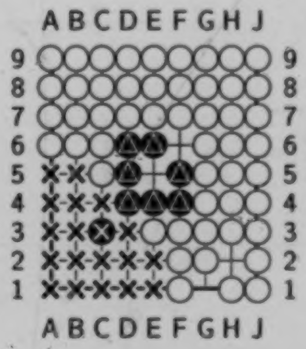
oneeyed.7.sgf



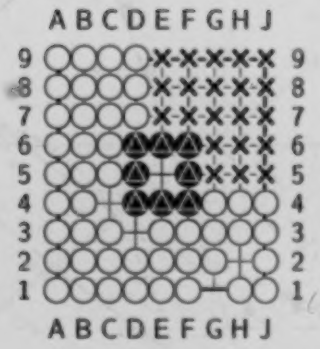
oneeyed.98sgf



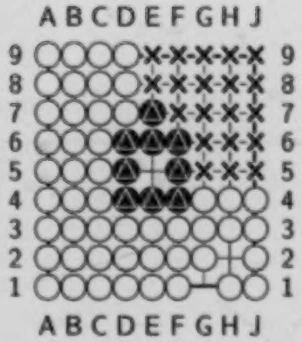
oneeyed.9.sgf



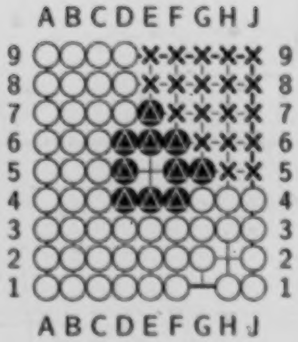
oneeyee.1.sgf



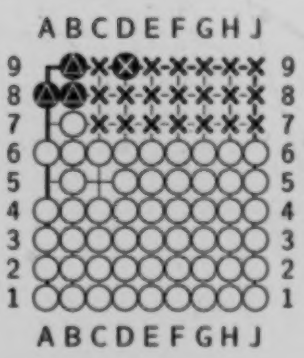
oneeyee.2.sgf



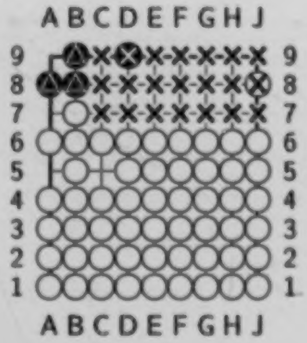
oneeyee.3.sgf



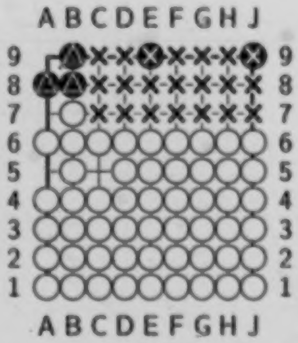
oneeyee.4.sgf



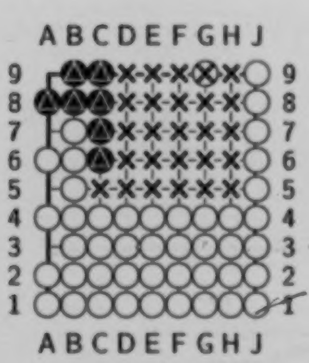
oneeyee.5.sgf



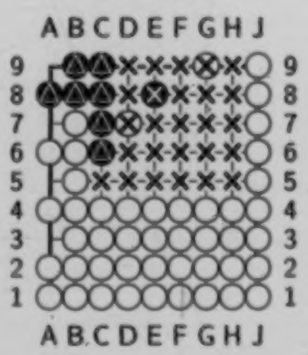
oneeyee.6.sgf



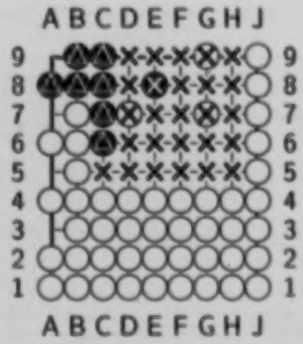
oneeyee.7.sgf



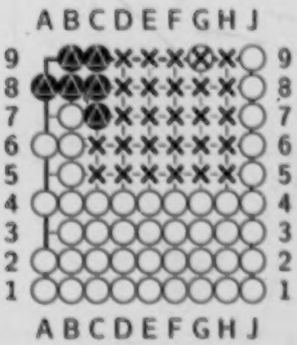
oneeyee.8.sgf



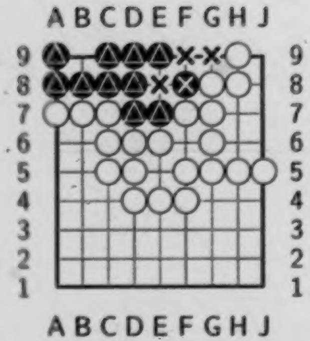
oneeyee.9.sgf



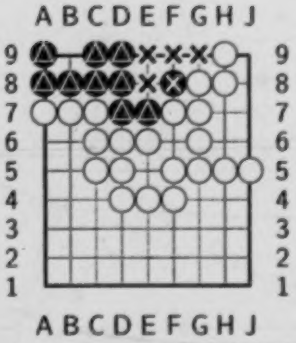
oneeyee.10.sgf



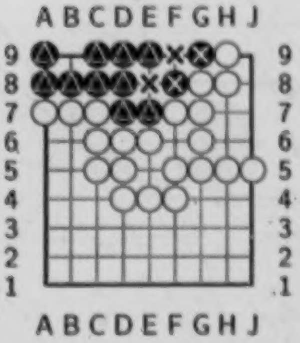
oneeyee.11.sgf



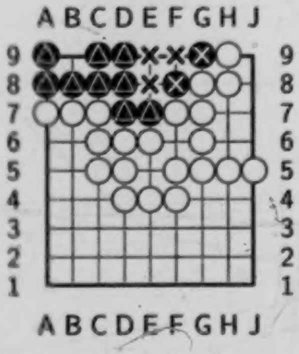
koeye.1.sgf



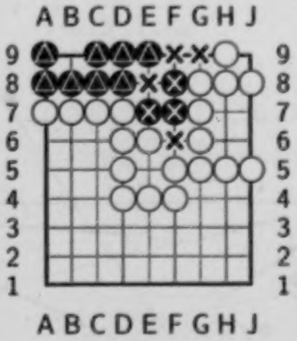
koeye.2.sgf



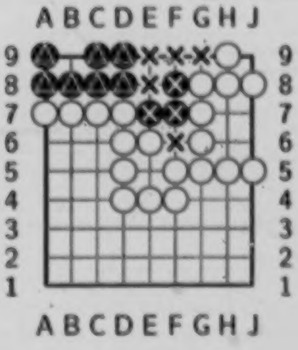
koeye.3.sgf



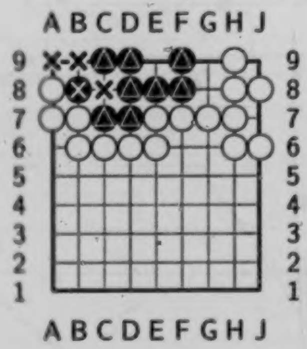
koeye.4.sgf



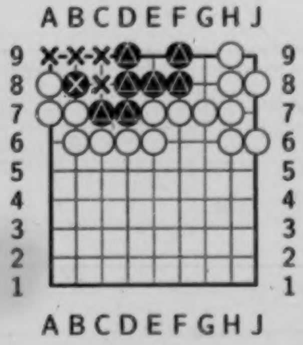
koeye.5.sgf



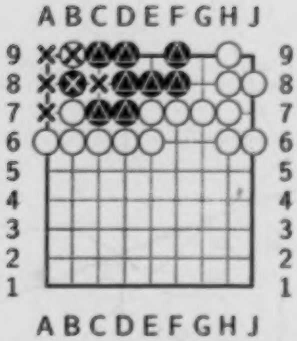
koeye.6.sgf



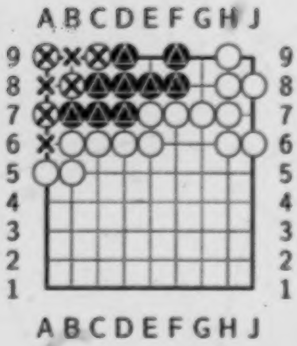
koeye.7.sgf



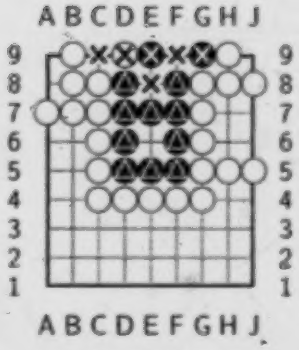
koeye.8.sgf



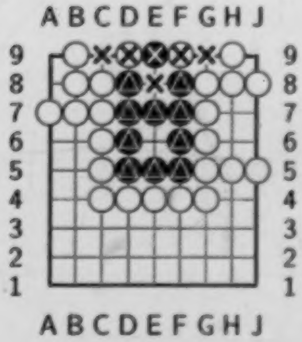
koeye.9.sgf



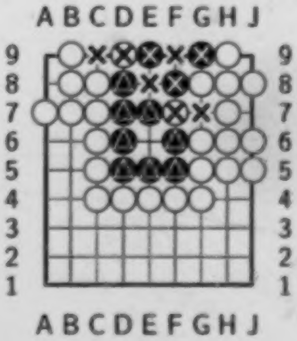
koeye.10.sgf



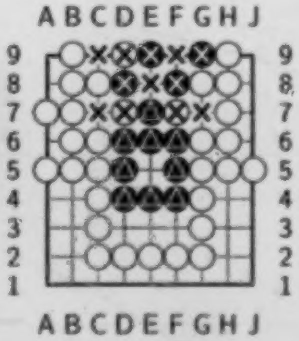
koeyeb.1.sgf



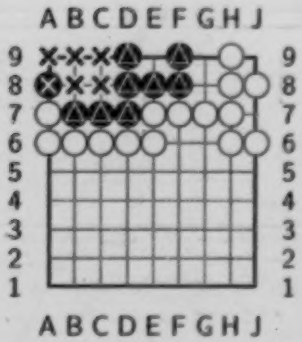
koeyeb.2.sgf



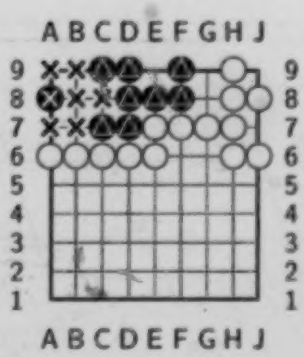
koeyeb.3.sgf



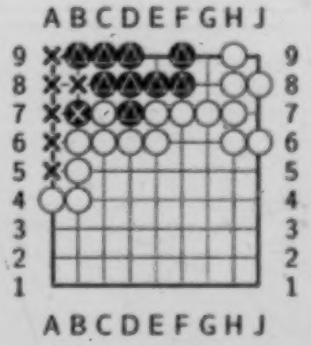
koeyeb.4.sgf



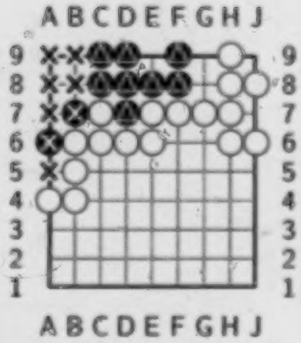
koeyeb.5.sgf



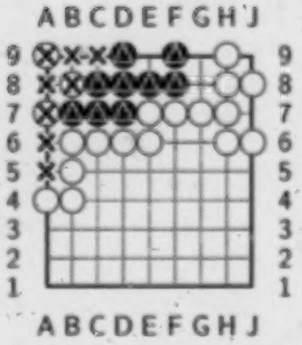
koeyeb.6.sgf



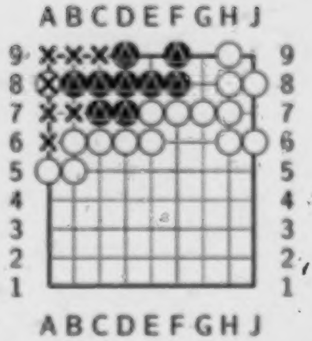
koeyeb.7.sgf



koeyeb.8.sgf

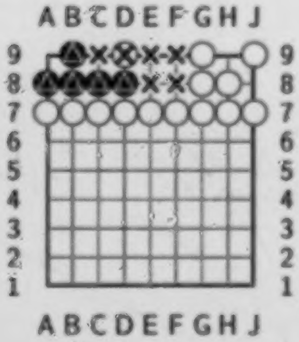


koeyeb.9.sgf

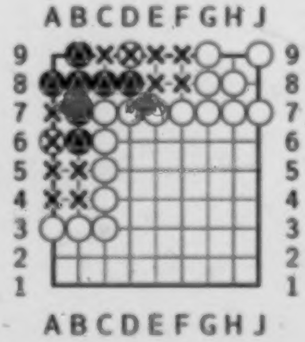


koeyeb.10.sgf

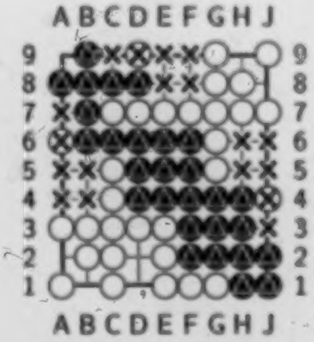
C.1.2 Positions Used in Chapter 7



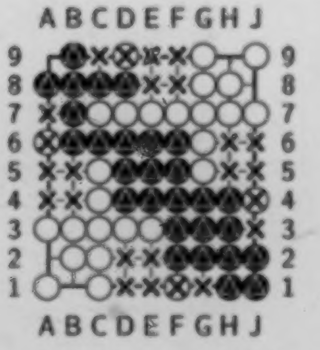
divide-conquer.1.sgf



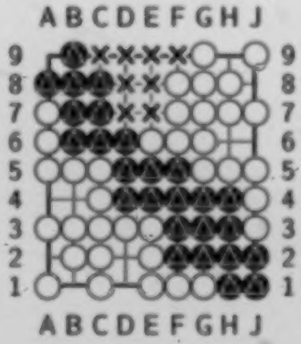
divide-conquer.2.sgf



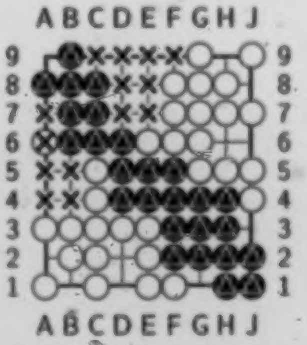
divide-conquer.3.sgf



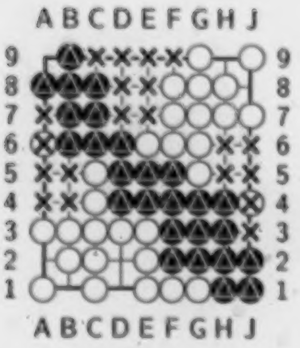
divide-conquer.4.sgf



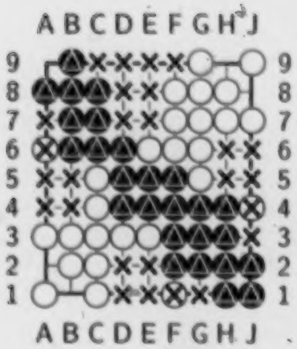
divide-conquer.5.sgf



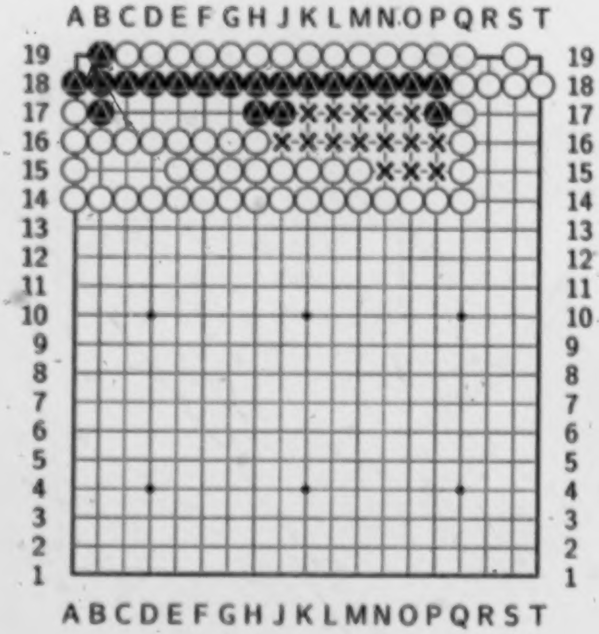
divide-conquer.6.sgf



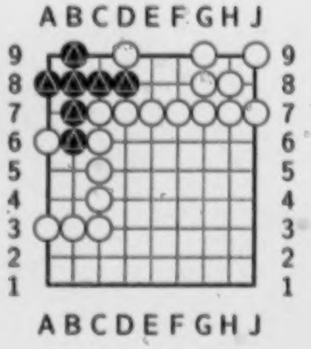
divide-conquer.7.sgf



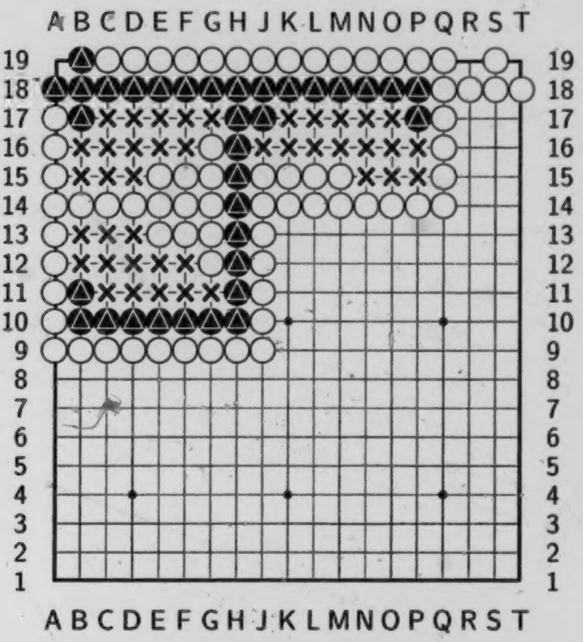
divide-conquer.8.sgf



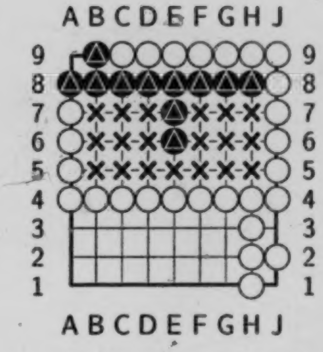
divide-conquer.9.sgf



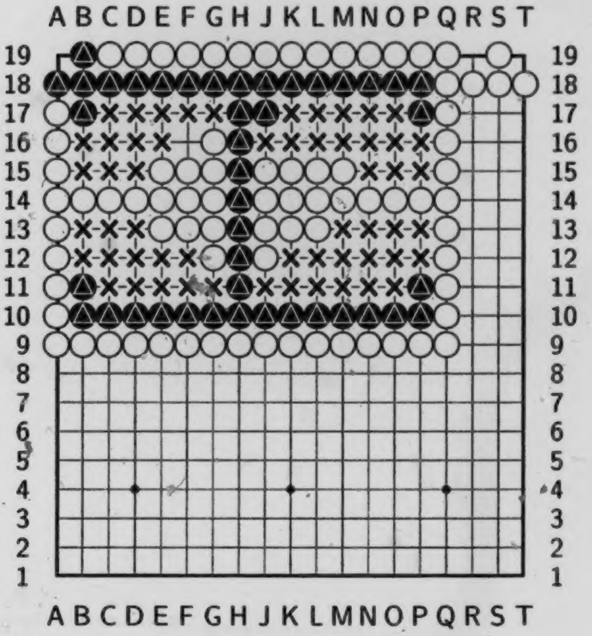
divide-conquer.10.sgf



divide-conquer.11.sgf

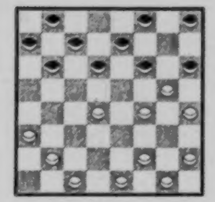


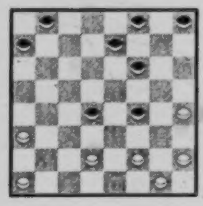
divide-conquer.13.sgf

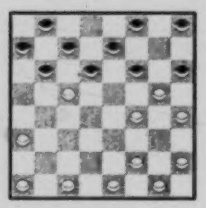


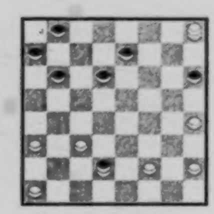
divide-conquer.12.sgf

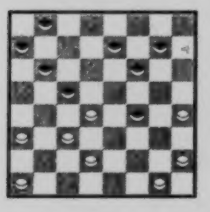
C.2 Checkers

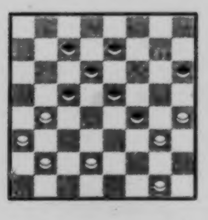

1 Black to play

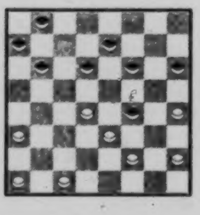

2 Black to play

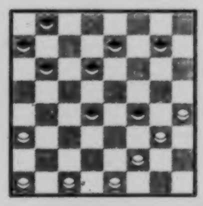

3 Black to play

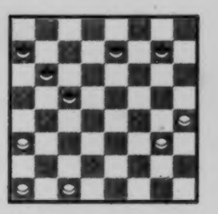

4 Black to play

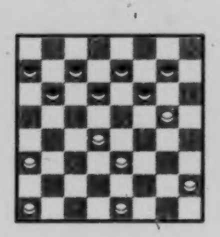

5 Black to play

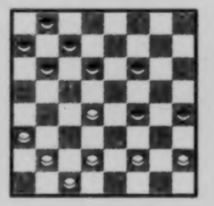

6 Black to play

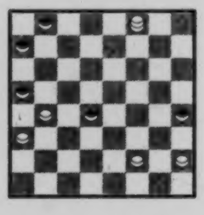

7 Black to play

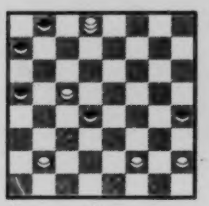

8 Black to play

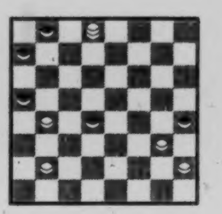

9 White to play

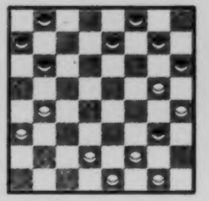

10 White to play

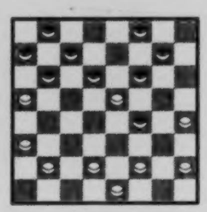

11 White to play

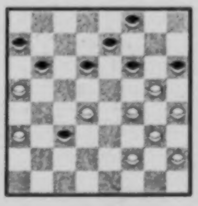

12 White to play


13 Black to play

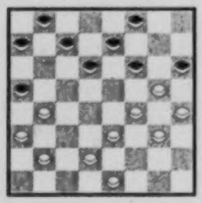

14 Black to play


15 Black to play

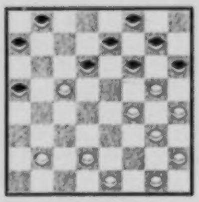

16 Black to play



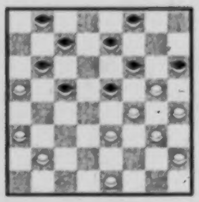
17 White to play



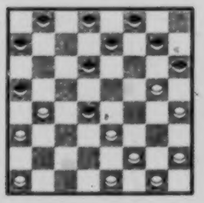
18 White to play



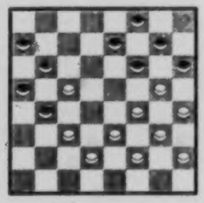
19 White to play



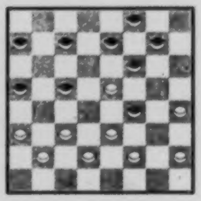
20 White to play



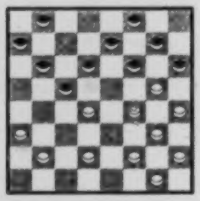
21 Black to play



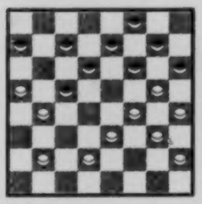
22 Black to play



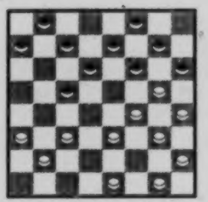
23 Black to play



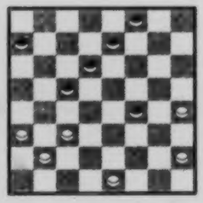
24 Black to play



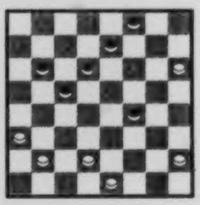
25 Black to play



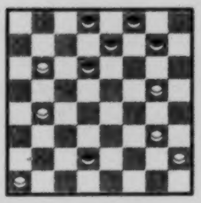
26 Black to play



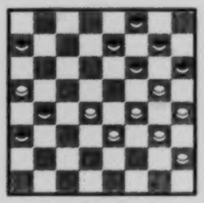
27 Black to play



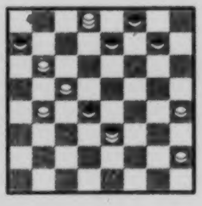
28 Black to play



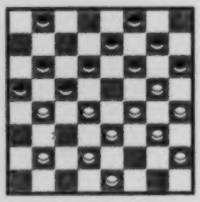
29 Black to play



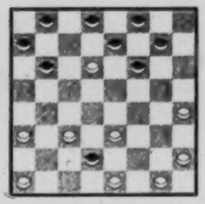
30 Black to play



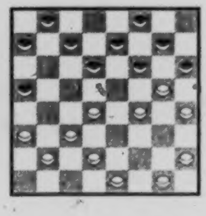
31 Black to play



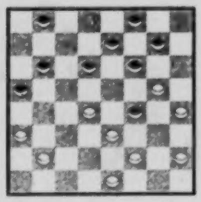
32 Black to play



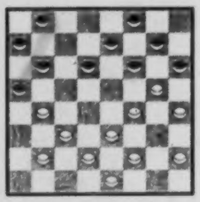
33 Black to play



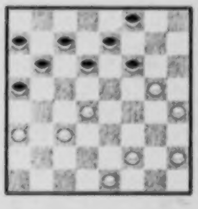
34 Black to play



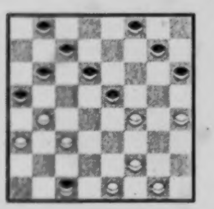
35 Black to play



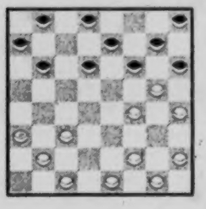
36 Black to play



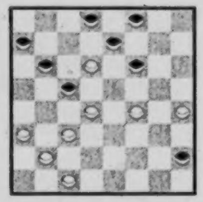
37 White to play



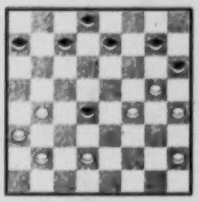
38 White to play



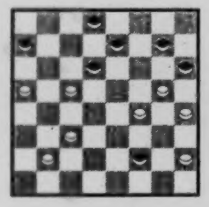
39 White to play



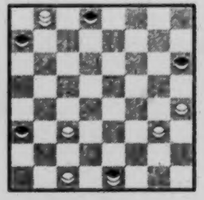
40 White to play



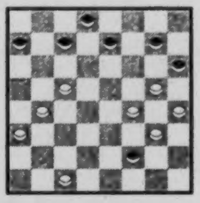
41 White to play



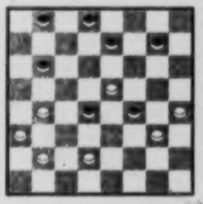
42 White to play



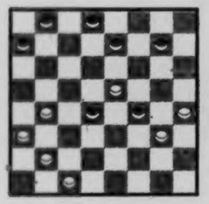
43 White to play



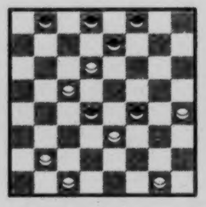
44 White to play



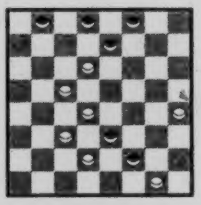
45 Black to play



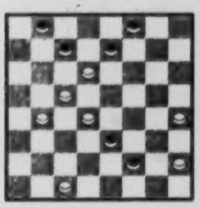
46 Black to play



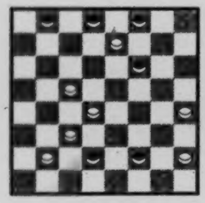
47 Black to play



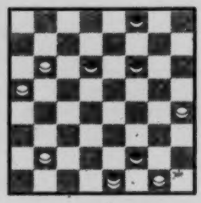
48 Black to play



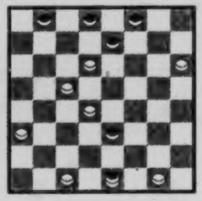
49 Black to play



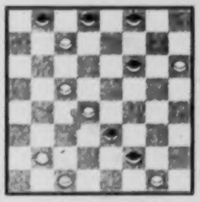
50 Black to play



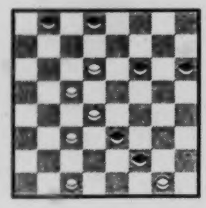
51 Black to play



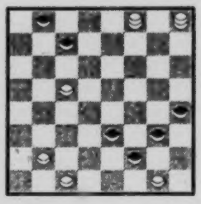
52 Black to play



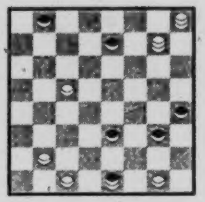
53 Black to play



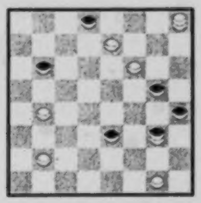
54 Black to play



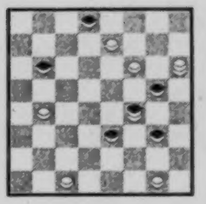
55 Black to play



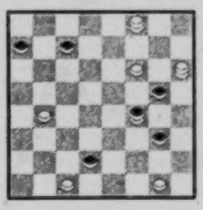
56 Black to play



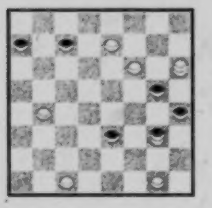
57 Black to play



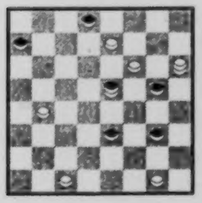
58 Black to play



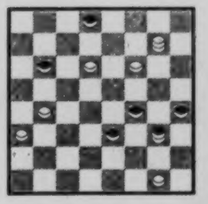
59 Black to play



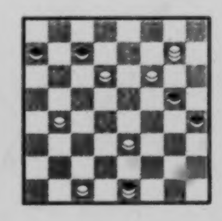
60 Black to play



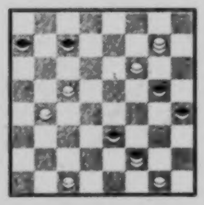
61 White to play



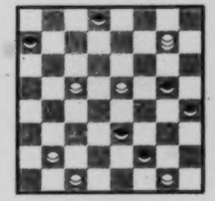
62 White to play



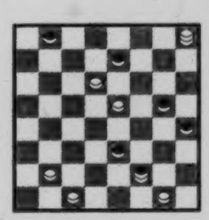
63 White to play



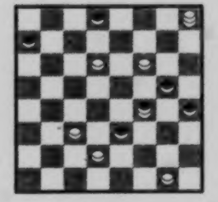
64 White to play



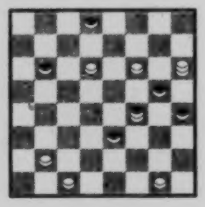
65 White to play



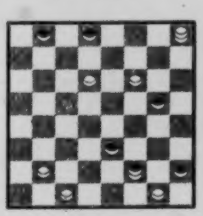
66 White to play



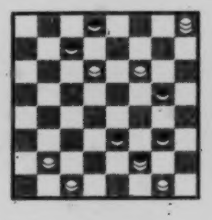
67 White to play



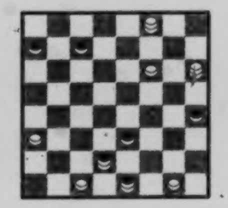
68 White to play



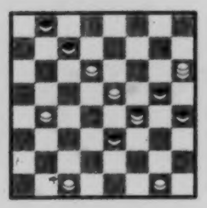
69 White to play



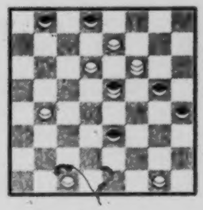
70 White to play



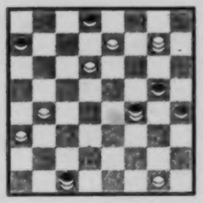
71 White to play



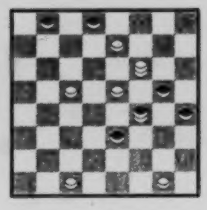
72 White to play



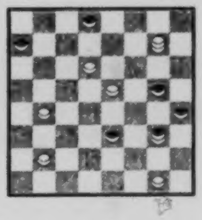
73 White to play



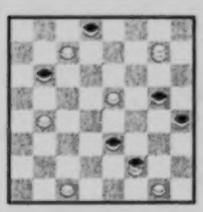
74 White to play



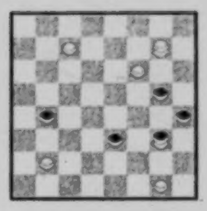
75 White to play



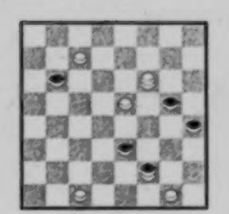
76 White to play



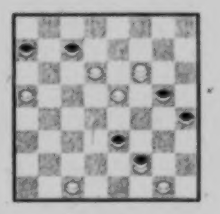
77 Black to play



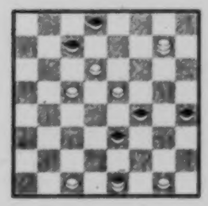
78 Black to play



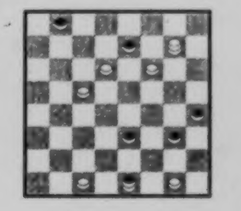
79 Black to play



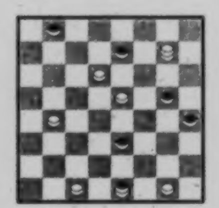
80 Black to play



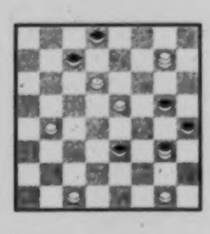
81 White to play



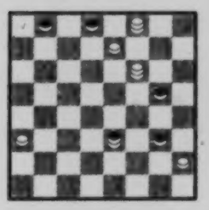
82 White to play



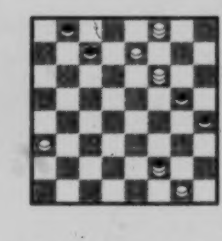
83 White to play



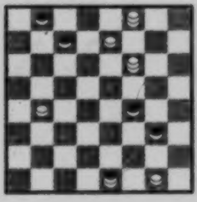
84 White to play



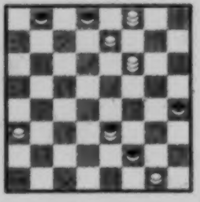
85 White to play



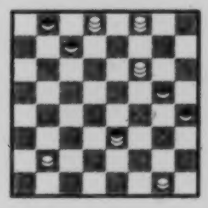
86 White to play



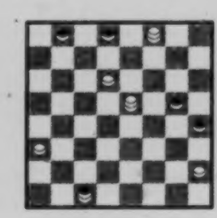
87 White to play



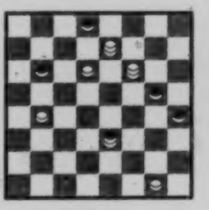
88 White to play



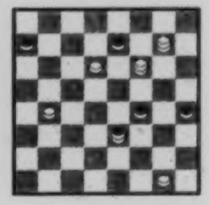
89 Black to play



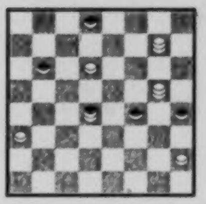
90 Black to play



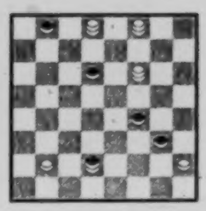
91 Black to play



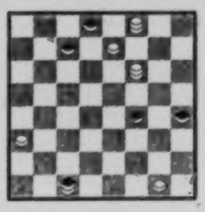
92 Black to play



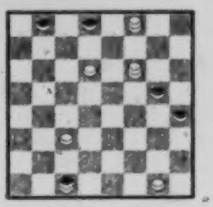
93 White to play



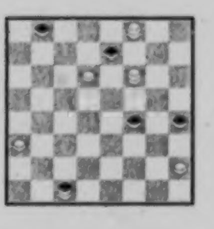
94 White to play

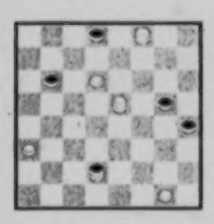


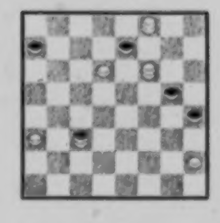
95 White to play

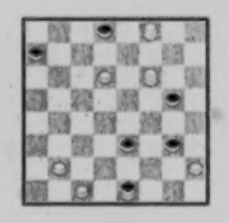


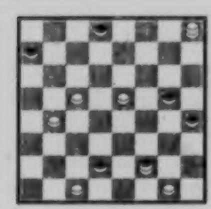
96 White to play

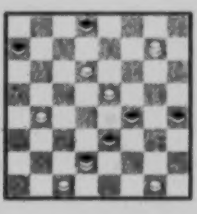

97 White to play

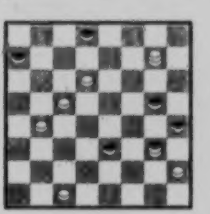

98 White to play

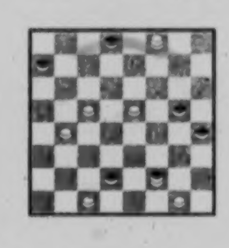

99 White to play

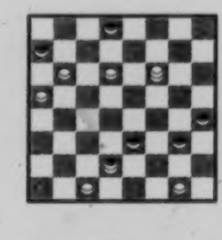

100 White to play

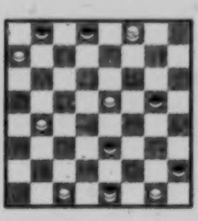

101 White to play

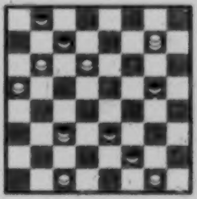

102 White to play

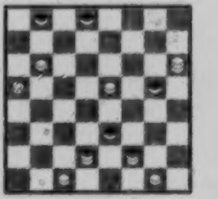

103 White to play

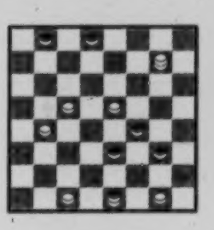

104 White to play

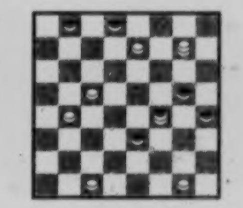

105 White to play

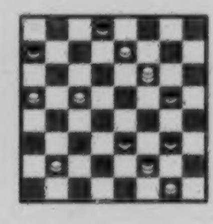

106 White to play

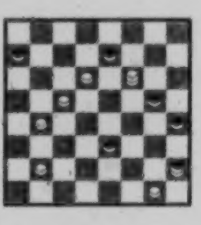

107 White to play

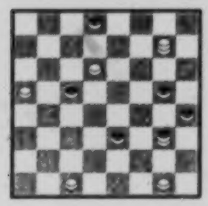

108 White to play

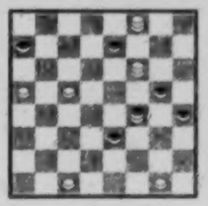

109 White to play

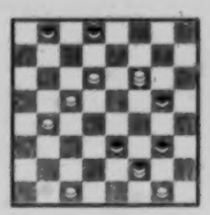

110 White to play

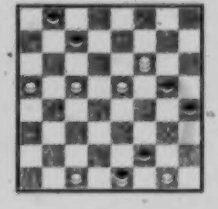

111 White to play

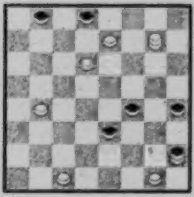

112 White to play

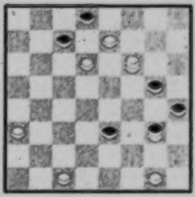

113 Black to play

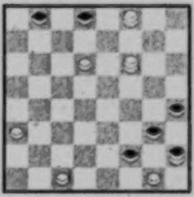

114 Black to play

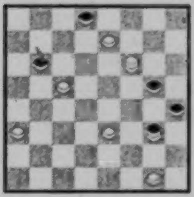

115 Black to play

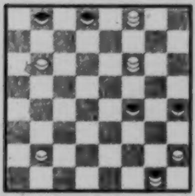

116 Black to play

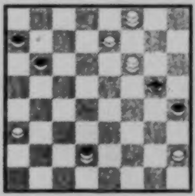

117 Black to play

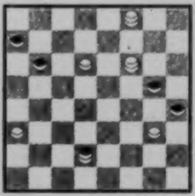

118 Black to play

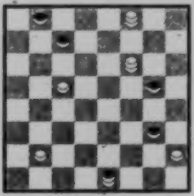

119 Black to play

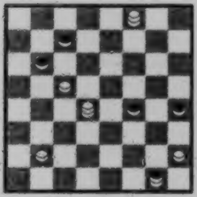

120 Black to play

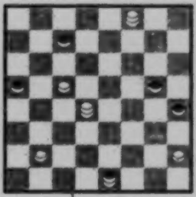

121 White to play

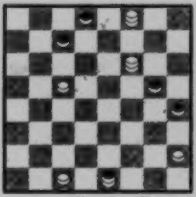

122 White to play

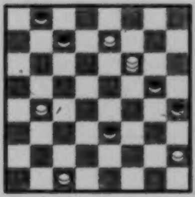

123 White to play

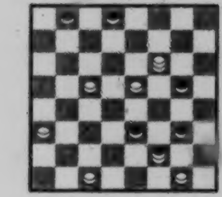

124 White to play

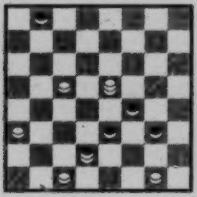

125 White to play

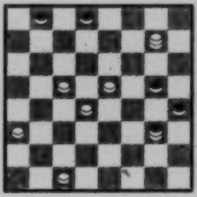

126 White to play

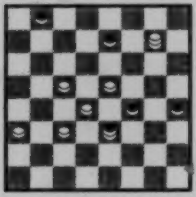

127 White to play

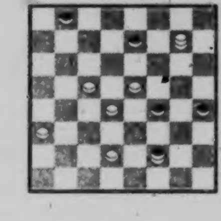

128 White to play

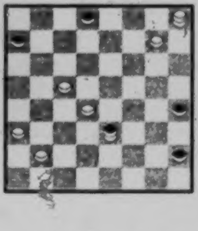

129 White to play

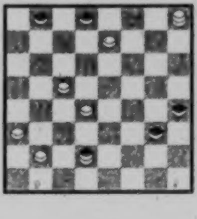

130 White to play

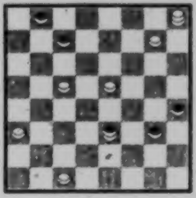

131 White to play

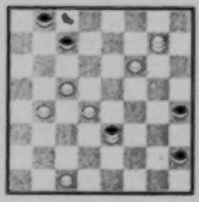

132 White to play

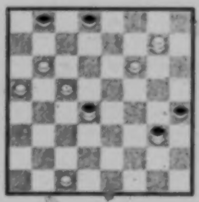

133 White to play

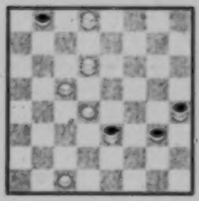

134 White to play

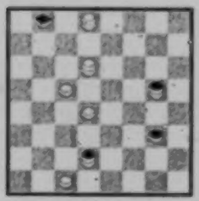

135 White to play

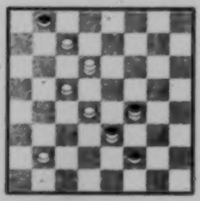

136 White to play

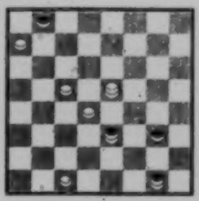

137 White to play

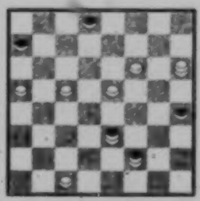

138 White to play

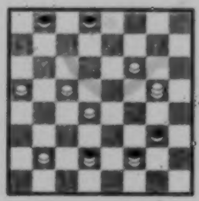

139 White to play

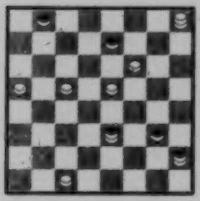

140 White to play

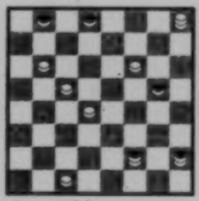

141 Black to play

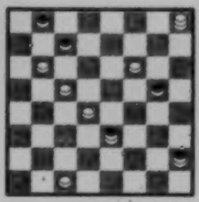

142 Black to play

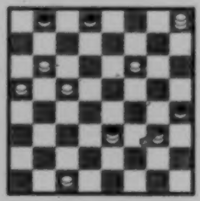

143 Black to play

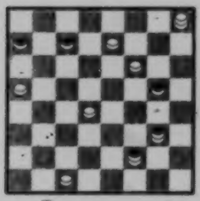

144 Black to play

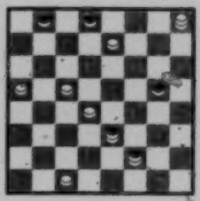

145 White to play

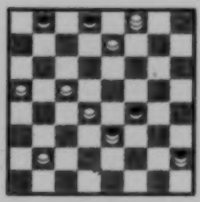

146 White to play

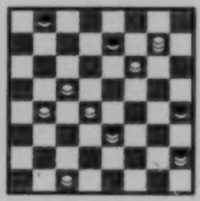

147 White to play

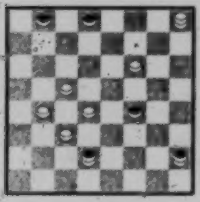

148 White to play

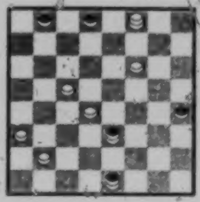

149 White to play

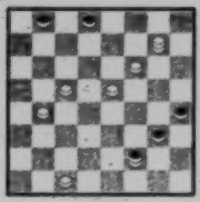

150 White to play

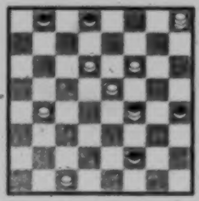

151 White to play

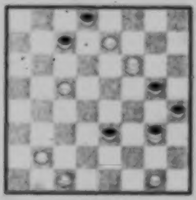

152 White to play


153 Black to play

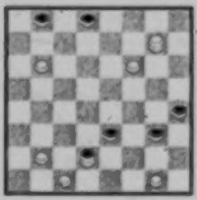

154 Black to play


155 Black to play

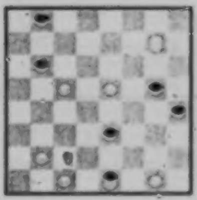

156 Black to play



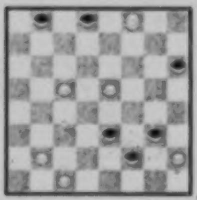
157 White to play



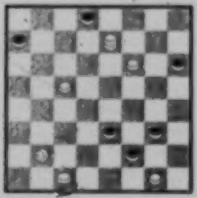
158 White to play



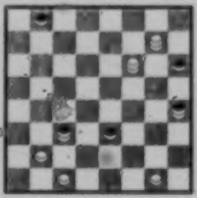
159 White to play



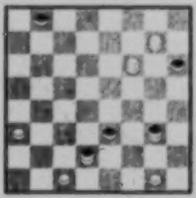
160 White to play



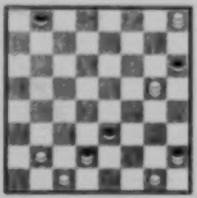
161 Black to play



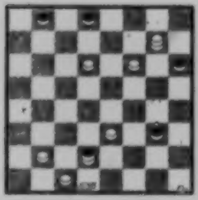
162 Black to play



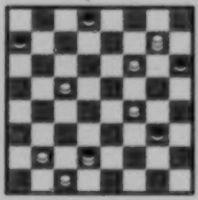
163 Black to play



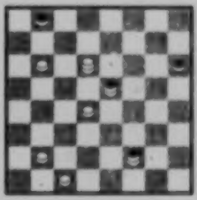
164 Black to play



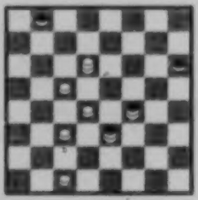
165 Black to play



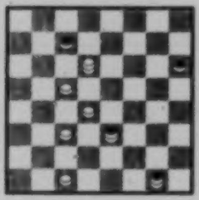
166 Black to play



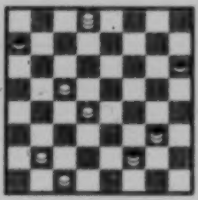
167 Black to play



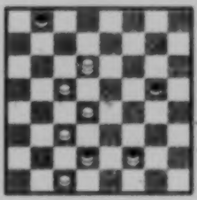
168 Black to play



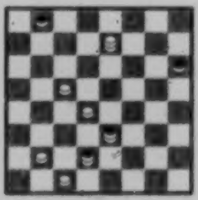
169 White to play



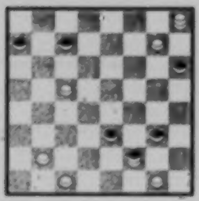
170 White to play



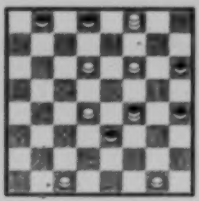
171 White to play



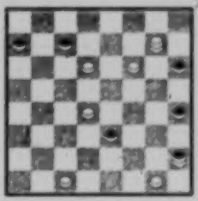
172 White to play



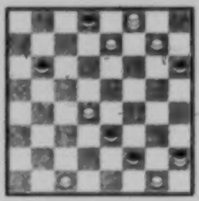
173 White to play



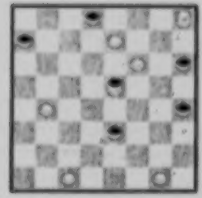
174 White to play

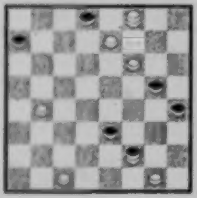


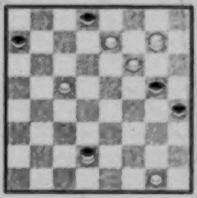
175 White to play

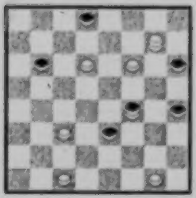


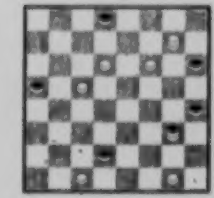
176 White to play

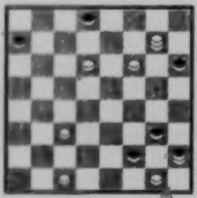

177 White to play

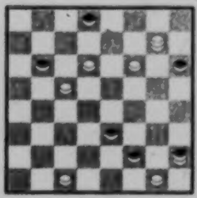

178 White to play

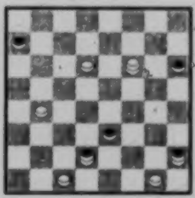

179 White to play

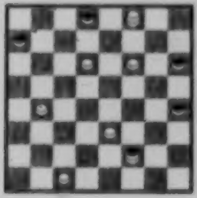

180 White to play

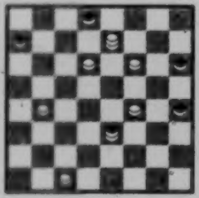

181 White to play

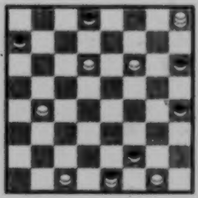

182 White to play

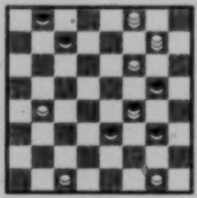

183 White to play

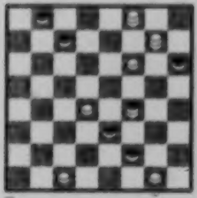

184 White to play

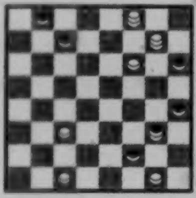

185 White to play

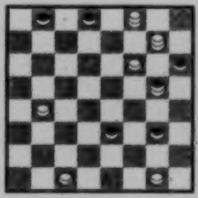

186 White to play

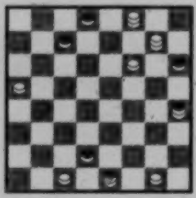

187 White to play

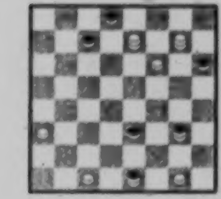

188 White to play

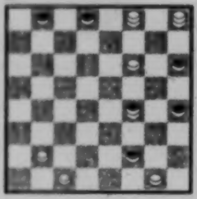

189 Black to play

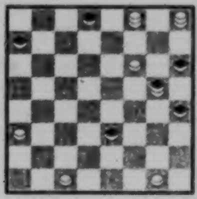

190 Black to play

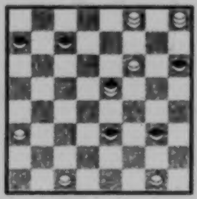

191 Black to play

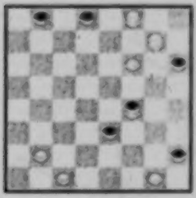

192 Black to play

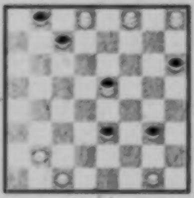

193 Black to play

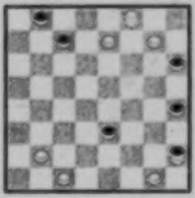

194 Black to play

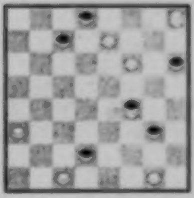

195 Black to play


196 Black to play


197 White to play


198 White to play


199 White to play


200 White to play